

THIRD EDITION

Join the discussion @ p2p.wrox.com



Professional

C++

Marc Gregoire

www.allitebooks.com

PROFESSIONAL C++

INTRODUCTION xli

► PART I INTRODUCTION TO PROFESSIONAL C++

CHAPTER 1	A Crash Course in C++ and the STL	3
CHAPTER 2	Working with Strings	47
CHAPTER 3	Coding with Style	57

► PART II PROFESSIONAL C++ SOFTWARE DESIGN

CHAPTER 4	Designing Professional C++ Programs	79
CHAPTER 5	Designing with Objects	107
CHAPTER 6	Designing for Reuse	127

► PART III CODING THE PROFESSIONAL WAY

CHAPTER 7	Gaining Proficiency with Classes and Objects.....	143
CHAPTER 8	Mastering Classes and Objects	177
CHAPTER 9	Discovering Inheritance Techniques.....	217
CHAPTER 10	C++ Quirks, Oddities, and Incidentals.....	273
CHAPTER 11	Writing Generic Code with Templates.....	315
CHAPTER 12	Demystifying C++ I/O	345
CHAPTER 13	Handling Errors.....	369
CHAPTER 14	Overloading C++ Operators	411
CHAPTER 15	Overview of the C++ Standard Library	443
CHAPTER 16	Understanding Containers and Iterators	467
CHAPTER 17	Mastering STL Algorithms	531
CHAPTER 18	String Localization and Regular Expressions.....	575
CHAPTER 19	Additional Library Utilities	601

► PART IV MASTERING ADVANCED FEATURES OF C++

CHAPTER 20	Customizing and Extending the STL	627
CHAPTER 21	Advanced Templates	673
CHAPTER 22	Memory Management	707
CHAPTER 23	Multithreaded Programming with C++	741

► PART V C++ SOFTWARE ENGINEERING

CHAPTER 24	Maximizing Software Engineering Methods	781
CHAPTER 25	Writing Efficient C++	801
CHAPTER 26	Conquering Debugging	827

APPENDIX A	C++ Interviews	863
APPENDIX B	Annotated Bibliography	885
APPENDIX C	Standard Library Header Files	895

INDEX	903
-------	-----

Professional C++

PROFESSIONAL
C++
Third Edition

Marc Gregoire



Professional C++, Third Edition

Published by

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-85805-9

ISBN: 978-1-118-85806-6 (ebk)

ISBN: 978-1-118-85813-4 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

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 <http://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 <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2014941048

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. [Insert third-party trademark information] 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 parents, who are always there for me.

—MARC GREGOIRE

ABOUT THE AUTHOR



MARC GREGOIRE is a software engineer from Belgium. He graduated from the University of Leuven, Belgium, with a degree in “Burgerlijk ingenieur in de computer wetenschappen” (equivalent to master of science in engineering in computer science). The year after, he received the cum laude degree of master in artificial intelligence 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 USA to EMEA and Asia. Now, Marc is working for Nikon Metrology on 3D laser scanning software.

His main expertise is C/C++, and specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24x7 on Windows and Linux platforms; for example, KNX/EIB home automation software. Next to C/C++, Marc also likes C# and uses PHP for creating web pages.

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

Marc is the founder of the Belgian C++ Users Group (www.becpp.org), and a member on the CodeGuru forum (as Marc G). He maintains a blog on www.nuonsoft.com/blog/, and is passionate about traveling and gastronomic restaurants.

ABOUT THE TECHNICAL EDITOR

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

He received his master of science in computer science summa cum laude with congratulations of the Board of Examiners from the University of Leuven, Belgium. In 2010, he completed his PhD at the declarative languages and artificial intelligence research group of the same university. His PhD research revolved around the efficient compilation of rule based programming languages, mainly to Java. During his doctoral studies, he was a teaching assistant for courses on object-oriented analysis and design, Java programming, and declarative programming languages.

Since graduating, Peter has worked for Nikon Metrology on 3D laser scanning and point cloud inspection software, in C++. In his spare time, he has co-authored two award-winning Windows 8 apps.

CREDITS

EXECUTIVE EDITOR
Robert Elliott

PROJECT EDITOR
Adaobi Obi Tulton

TECHNICAL EDITOR
Peter Van Weert

PRODUCTION EDITOR
Christine Mugnolo

COPY EDITOR
Mike La Bonne

**MANAGER OF CONTENT DEVELOPMENT
AND ASSEMBLY**
Mary Beth Wakefield

DIRECTOR OF COMMUNITY MARKETING
David Mayhew

MARKETING MANAGER
Carrie Sherrill

BUSINESS MANAGER
Amy Knies

**VICE PRESIDENT AND EXECUTIVE GROUP
PUBLISHER**
Richard Swadley

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Todd Klemme

PROOFREADER
Jennifer Bennett, Word One New York

TECHNICAL PROOFREADER
Michael McLaughlin

INDEXER
Johnna Dinse

COVER DESIGNER
Wiley

COVER IMAGE
©iStock.com/PLAINVIEW

ACKNOWLEDGMENTS

I THANK THE JOHN WILEY & SONS and Wrox Press editorial and production teams for their support. Especially, thank you to Robert Elliott, executive editor at Wiley, for giving me a chance to write this new edition and Adaobi Obi Tulton, project editor, for managing this project.

I also thank the original authors of the first edition, Nicholas A. Solter and Scott J. Kleper, for doing such a great job on which I could base this new edition.

A very special thank you to my technical editor, Peter Van Weert, for his outstanding technical review. His many constructive comments and ideas have certainly made this book better. Also thank you to my technical proofreader, Michael B. McLaughlin, for doing a great job on proofreading the final manuscript.

Of course, the support and patience of my parents, my brother and his wife were very important in finishing this book.

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

CONTENTS

INTRODUCTION

xli

PART I: INTRODUCTION TO PROFESSIONAL C++

CHAPTER 1: A CRASH COURSE IN C++ AND THE STL	3
The Basics of C++	4
The Obligatory Hello, World	4
Comments	4
Preprocessor Directives	5
The main() Function	6
I/O Streams	6
Namespaces	7
Variables	9
Literals	11
Operators	11
Types	13
Enumerated Types	13
Structs	15
Conditionals	15
if/else Statements	16
switch Statements	16
The Conditional Operator	17
Logical Evaluation Operators	17
Arrays	18
std::array	20
Loops	20
The while Loop	20
The do/while Loop	21
The for Loop	21
The Range-Based for Loop	21
Functions	22
Alternative Function Syntax	23
Function Return Type Deduction	23
Type Inference Part 1	24
The auto Keyword	24
The decltype Keyword	24

Those Are the Basics	24
Diving Deeper into C++	25
Pointers and Dynamic Memory	25
The Stack and the Heap	25
Working with Pointers	26
Dynamically Allocated Arrays	27
Null Pointer Constant	28
Smart Pointers	28
References	29
Strings in C++	30
Exceptions	30
The Many Uses of const	32
const Constants	32
const to Protect Parameters	32
const References	32
Type Inference Part 2	33
decltype(auto)	33
C++ as an Object-Oriented Language	34
Defining a Class	34
The Standard Library	36
std::vector	36
Your First Useful C++ Program	37
An Employee Records System	37
The Employee Class	38
Employee.h	38
Employee.cpp	39
EmployeeTest.cpp	41
The Database Class	41
Database.h	41
Database.cpp	42
DatabaseTest.cpp	43
The User Interface	44
UserInterface.cpp	44
Evaluating the Program	46
Summary	46
CHAPTER 2: WORKING WITH STRINGS	47
Dynamic Strings	47
C-Style Strings	48
String Literals	50
The C++ string Class	51

What Is Wrong with C-Style Strings?	51
Using the <code>string</code> Class	51
<code>std::string</code> Literals	53
Numeric Conversions	53
Raw String Literals	54
Nonstandard Strings	55
Summary	55
CHAPTER 3: CODING WITH STYLE	57
The Importance of Looking Good	57
Thinking Ahead	58
Elements of Good Style	58
Documenting Your Code	58
Reasons to Write Comments	58
Commenting to Explain Usage	58
Commenting to Explain Complicated Code	60
Commenting to Convey Metainformation	61
Commenting Styles	62
Commenting Every Line	62
Prefix Comments	64
Fixed-Format Comments	64
Ad Hoc Comments	65
Self-Documenting Code	66
Comments in This Book	66
Decomposition	66
Decomposition through Refactoring	67
Decomposition by Design	67
Decomposition in This Book	68
Naming	68
Choosing a Good Name	68
Naming Conventions	69
Counters	69
Prefixes	69
Getters and Setters	70
Capitalization	70
Namespaced Constants	70
Hungarian Notation	70
Using Language Features with Style	71
Use Constants	71
Use References Instead of Pointers	71
Use Custom Exceptions	72

Formatting	72
The Curly Brace Alignment Debate	72
Coming to Blows over Spaces and Parentheses	74
Spaces and Tabs	74
Stylistic Challenges	74
Summary	75

PART II: PROFESSIONAL C++ SOFTWARE DESIGN

CHAPTER 4: DESIGNING PROFESSIONAL C++ PROGRAMS	79
What Is Programming Design?	80
The Importance of Programming Design	80
Designing for C++	82
Two Rules for C++ Design	83
Abstraction	84
Benefiting from Abstraction	84
Incorporating Abstraction in Your Design	84
Reuse	85
Reusing Code	86
Writing Reusable Code	86
Reusing Ideas	87
Reusing Code	87
A Note on Terminology	87
Deciding Whether or Not to Reuse Code	88
Advantages to Reusing Code	88
Disadvantages to Reusing Code	89
Putting It Together to Make a Decision	90
Strategies for Reusing Code	90
Understand the Capabilities and Limitations	90
Understand the Performance	91
Understand Platform Limitations	94
Understand Licensing and Support	94
Know Where to Find Help	94
Prototype	95
Bundling Third-Party Applications	95
Open-Source Libraries	96
The Open-Source Movements	96
Finding and Using Open-Source Libraries	96
Guidelines for Using Open-Source Code	97
The C++ Standard Library	97
C Standard Library	97

Deciding Whether or Not to Use the STL	98
Designing with Patterns and Techniques	98
Designing a Chess Program	99
Requirements	99
Design Steps	99
Divide the Program into Subsystems	99
Choose Threading Models	101
Specify Class Hierarchies for Each Subsystem	101
Specify Classes, Data Structures, Algorithms, and Patterns for Each Subsystem	102
Specify Error Handling for Each Subsystem	104
Summary	105
CHAPTER 5: DESIGNING WITH OBJECTS	107
Am I Thinking Procedurally?	108
The Object-Oriented Philosophy	108
Classes	108
Components	109
Properties	109
Behaviors	110
Bringing It All Together	110
Living in a World of Objects	111
Overobjectification	111
Overly General Objects	112
Object Relationships	113
The Has-A Relationship	113
The Is-A Relationship (Inheritance)	114
Inheritance Techniques	115
Polymorphism versus Code Reuse	116
The Fine Line between Has-A and Is-A	116
The Not-A Relationship	119
Hierarchies	120
Multiple Inheritance	121
Mixin Classes	122
Abstraction	122
Interface versus Implementation	123
Deciding on an Exposed Interface	123
Consider the Audience	123
Consider the Purpose	124
Consider the Future	125
Designing a Successful Abstraction	125
Summary	126

CHAPTER 6: DESIGNING FOR REUSE	127
The Reuse Philosophy	127
How to Design Reusable Code	128
Use Abstraction	129
Structure Your Code for Optimal Reuse	130
Avoid Combining Unrelated or Logically Separate Concepts	130
Use Templates for Generic Data Structures and Algorithms	132
Provide Appropriate Checks and Safeguards	134
Design Usable Interfaces	135
Design Interfaces That Are Easy to Use	135
Design General-Purpose Interfaces	138
Reconciling Generality and Ease of Use	139
Supply Multiple Interfaces	139
Make Common Functionality Easy to Use	140
Summary	140
PART III: CODING THE PROFESSIONAL WAY	
CHAPTER 7: GAINING PROFICIENCY WITH CLASSES AND OBJECTS	143
Introducing the Spreadsheet Example	144
Writing Classes	144
Class Definitions	144
Class Members	145
Access Control	145
Order of Declarations	147
Defining Methods	147
Accessing Data Members	148
Calling Other Methods	148
The this Pointer	150
Using Objects	151
Objects on the Stack	151
Objects on the Heap	151
Object Life Cycles	152
Object Creation	153
Writing Constructors	153
Using Constructors	154
Providing Multiple Constructors	155
Default Constructors	156
Constructor Initializers	160

Copy Constructors	162
Initializer-List Constructors	165
In-Class Member Initializers	167
Delegating Constructors	167
Summary of Compiler-Generated Constructors	168
Object Destruction	169
Assigning to Objects	170
Declaring an Assignment Operator	171
Defining an Assignment Operator	172
Explicitly Defaulted and Deleted Assignment Operator	173
Distinguishing Copying from Assignment	173
Objects as Return Values	173
Copy Constructors and Object Members	174
Summary	175
CHAPTER 8: MASTERING CLASSES AND OBJECTS	177
Dynamic Memory Allocation in Objects	178
The Spreadsheet Class	178
Freeing Memory with Destructors	180
Handling Copying and Assignment	180
The Spreadsheet Copy Constructor	185
The Spreadsheet Assignment Operator	185
Common Helper Routines for Copy Constructor and Assignment Operator	187
Disallowing Assignment and Pass-By-Value	188
Different Kinds of Data Members	188
static Data Members	188
Accessing static Data Members within Class Methods	189
Accessing static Data Members Outside Methods	190
const Data Members	190
Reference Data Members	191
const Reference Data Members	192
More about Methods	193
static Methods	193
const Methods	194
mutable Data Members	195
Method Overloading	195
Default Parameters	197
Inline Methods	197
Nested Classes	199
Enumerated Types Inside Classes	201

Friends	202
Operator Overloading	203
Example: Implementing Addition for SpreadsheetCells	203
First Attempt: The add Method	203
Second Attempt: Overloaded operator+ as a Method	204
Third Attempt: Global operator+	206
Overloading Arithmetic Operators	207
Overloading the Arithmetic Shorthand Operators	208
Overloading Comparison Operators	210
Building Types with Operator Overloading	211
Building Stable Interfaces	212
Using Interface and Implementation Classes	212
Summary	215
 CHAPTER 9: DISCOVERING INHERITANCE TECHNIQUES	 217
Building Classes with Inheritance	218
Extending Classes	218
A Client's View of Inheritance	219
A Derived Class's View of Inheritance	220
Preventing Inheritance	221
Overriding Methods	222
How I Learned to Stop Worrying and Make Everything virtual	222
Syntax for Overriding a Method	223
A Client's View of Overridden Methods	223
Preventing Overriding	225
Inheritance for Reuse	225
The WeatherPrediction Class	225
Adding Functionality in a Derived Class	226
Replacing Functionality in a Derived Class	228
Respect Your Parents	229
Parent Constructors	229
Parent Destructors	230
Referring to Parent Names	232
Casting Up and Down	235
Inheritance for Polymorphism	236
Return of the Spreadsheet	236
Designing the Polymorphic Spreadsheet Cell	237
The Spreadsheet Cell Base Class	237
A First Attempt	238
Pure Virtual Methods and Abstract Base Classes	238
Base Class Source Code	239

The Individual Derived Classes	239
String Spreadsheet Cell Class Definition	240
String Spreadsheet Cell Implementation	240
Double Spreadsheet Cell Class Definition and Implementation	241
Leveraging Polymorphism	242
Future Considerations	243
Multiple Inheritance	244
Inheriting from Multiple Classes	244
Naming Collisions and Ambiguous Base Classes	246
Name Ambiguity	246
Ambiguous Base Classes	247
Uses for Multiple Inheritance	248
Interesting and Obscure Inheritance Issues	249
Changing the Overridden Method's Characteristics	249
Changing the Method Return Type	249
Changing the Method Parameters	250
Inherited Constructors	253
Special Cases in Overriding Methods	256
The Base Class Method Is static	257
The Base Class Method Is Overloaded	258
The Base Class Method Is private or protected	259
The Base Class Method Has Default Arguments	260
The Base Class Method Has a Different Access Level	261
Copy Constructors and Assignment Operator in Derived Classes	264
The Truth about virtual	265
Hiding Instead of Overriding	265
How virtual Is Implemented	266
The Justification for virtual	267
The Need for virtual Destructors	267
Run-Time Type Facilities	268
Non-Public Inheritance	270
Virtual Base Classes	270
Summary	271
CHAPTER 10: C++ QUIRKS, ODDITIES, AND INCIDENTALS	273
References	274
Reference Variables	274
Modifying References	275
References to Pointers and Pointers to References	275
Reference Data Members	276
Reference Parameters	276

References from Pointers	277
Pass-by-Reference Versus Pass-by-Value	277
Reference Return Values	278
Deciding between References and Pointers	278
Rvalue References	281
Move Semantics	283
Keyword Confusion	286
The <code>const</code> Keyword	286
<code>const</code> Variables and Parameters	287
<code>const</code> Methods	289
The <code>constexpr</code> Keyword	289
The <code>static</code> Keyword	291
<code>static</code> Data Members and Methods	291
<code>static</code> Linkage	291
<code>static</code> Variables in Functions	293
Order of Initialization of Nonlocal Variables	294
Order of Destruction of Nonlocal Variables	294
Types and Casts	295
<code>typedefs</code>	295
<code>typedefs</code> for Function Pointers	296
Type Aliases	297
Casts	297
<code>const_cast</code>	298
<code>static_cast</code>	298
<code>reinterpret_cast</code>	299
<code>dynamic_cast</code>	300
Summary of Casts	301
Scope Resolution	302
C++11 / C++14	303
Uniform Initialization	303
Initializer Lists	305
Explicit Conversion Operators	305
Attributes	306
User-Defined Literals	307
Standard User-Defined Literals	308
Header Files	309
C Utilities	310
Variable-Length Argument Lists	310
Accessing the Arguments	311
Why You Shouldn't Use C-Style Variable-Length Argument Lists	312
Preprocessor Macros	312
Summary	313

CHAPTER 11: WRITING GENERIC CODE WITH TEMPLATES	315
Overview of Templates	316
Class Templates	317
Writing a Class Template	317
Coding without Templates	317
A Template Grid Class	320
Using the Grid Template	323
Angle Brackets	324
How the Compiler Processes Templates	324
Selective Instantiation	325
Template Requirements on Types	325
Distributing Template Code between Files	325
Template Definitions in Header Files	325
Template Definitions in Source Files	326
Template Parameters	327
Non-Type Template Parameters	327
Default Values for Type Parameters	329
Method Templates	330
Method Templates with Non-Type Parameters	332
Class Template Specialization	334
Deriving from Class Templates	336
Inheritance versus Specialization	337
Alias Templates	338
Alternative Function Syntax	338
Function Templates	339
Function Template Specialization	340
Function Template Overloading	341
Function Template Overloading and Specialization Together	342
Friend Function Templates of Class Templates	342
Variable Templates	343
Summary	343
CHAPTER 12: DEMYSTIFYING C++ I/O	345
Using Streams	346
What Is a Stream, Anyway?	346
Stream Sources and Destinations	347
Output with Streams	347
Output Basics	347
Methods of Output Streams	348
Handling Output Errors	350
Output Manipulators	351

Input with Streams	353
Input Basics	353
Input Methods	354
Handling Input Errors	357
Input Manipulators	358
Input and Output with Objects	359
String Streams	360
File Streams	362
Jumping around with <code>seek()</code> and <code>tell()</code>	363
Linking Streams Together	365
Bidirectional I/O	366
Summary	367
CHAPTER 13: HANDLING ERRORS	369
Errors and Exceptions	370
What Are Exceptions, Anyway?	370
Why Exceptions in C++ Are a Good Thing	371
Recommendation	372
Exception Mechanics	372
Throwing and Catching Exceptions	373
Exception Types	376
Catching Exception Objects by <code>const</code> and Reference	377
Throwing and Catching Multiple Exceptions	378
Matching and <code>const</code>	379
Matching Any Exception	380
Uncaught Exceptions	380
Throw Lists	382
Unexpected Exceptions	383
Changing the Throw List in Overridden Methods	385
Are Throw Lists Useful?	386
Exceptions and Polymorphism	387
The Standard Exception Hierarchy	387
Catching Exceptions in a Class Hierarchy	388
Writing Your Own Exception Classes	390
Nested Exceptions	392
Stack Unwinding and Cleanup	394
Use Smart Pointers	396
Catch, Cleanup, and Rethrow	396
Common Error-Handling Issues	397
Memory Allocation Errors	397
Non-Throwing new	398

Customizing Memory Allocation Failure Behavior	399
Errors in Constructors	400
Function-Try-Blocks for Constructors	402
Errors in Destructors	404
Putting It All Together	405
Summary	409
CHAPTER 14: OVERLOADING C++ OPERATORS	411
Overview of Operator Overloading	412
Why Overload Operators?	412
Limitations to Operator Overloading	412
Choices in Operator Overloading	413
Method or Global Function	413
Choosing Argument Types	414
Choosing Return Types	414
Choosing Behavior	415
Operators You Shouldn't Overload	415
Summary of Overloadable Operators	415
Rvalue References	419
Relational Operators	419
Overloading the Arithmetic Operators	420
Overloading Unary Minus and Unary Plus	420
Overloading Increment and Decrement	420
Overloading the Bitwise and Binary Logical Operators	421
Overloading the Insertion and Extraction Operators	422
Overloading the Subscripting Operator	423
Providing Read-Only Access with operator[]	426
Non-Integral Array Indices	427
Overloading the Function Call Operator	428
Overloading the Dereferencing Operators	429
Implementing operator*	430
Implementing operator->	431
What in the World Is operator->* ?	432
Writing Conversion Operators	432
Ambiguity Problems with Conversion Operators	433
Conversions for Boolean Expressions	434
Overloading the Memory Allocation and Deallocation Operators	436
How new and delete Really Work	436
The New-Expression and operator new	437

The Delete-Expression and operator delete	437
Overloading operator new and operator delete	437
Explicitly Deleting/Defaulting operator new and operator delete	440
Overloading operator new and operator delete with Extra Parameters	440
Summary	442
CHAPTER 15: OVERVIEW OF THE C++ STANDARD LIBRARY	443
Coding Principles	444
Use of Templates	444
Use of Operator Overloading	444
Overview of the C++ Standard Library	445
Strings	445
Regular Expressions	445
I/O Streams	445
Smart Pointers	446
Exceptions	446
Mathematical Utilities	446
Time Utilities	447
Random Numbers	447
Initializer Lists	447
Pair and Tuple	447
Function Objects	448
Multithreading	448
Type Traits	448
The Standard Template Library	448
STL Containers	448
STL Algorithms	456
What's Missing from the STL	465
Summary	465
CHAPTER 16: UNDERSTANDING CONTAINERS AND ITERATORS	467
Containers Overview	468
Requirements on Elements	468
Exceptions and Error Checking	470
Iterators	470
Common Iterator typedefs and Methods	472
Sequential Containers	473

vector	473
vector Overview	473
vector Details	475
vector Example: A Round-Robin Class	486
The vector<bool> Specialization	490
deque	491
list	491
Accessing Elements	492
Iterators	492
Adding and Removing Elements	492
list Size	492
Special list Operations	492
forward_list	495
array	497
Container Adapters	498
queue	498
queue Operations	499
queue Example: A Network Packet Buffer	499
priority_queue	501
priority_queue Operations	502
priority_queue Example: An Error Correlator	502
stack	504
stack Operations	504
stack Example: Revised Error Correlator	504
Associative Containers	504
The pair Utility Class	504
map	505
Constructing maps	506
Inserting Elements	506
map Iterators	508
Looking Up Elements	509
Removing Elements	509
map Example: Bank Account	510
multimap	512
multimap Example: Buddy Lists	512
set	515
set Example: Access Control List	515
multiset	516
Unordered Associative Containers/Hash Tables	516
Hash Functions	517
unordered_map	519

unordered_map Example: Phone Book	521
unordered_multimap	522
unordered_set/unordered_multiset	522
Other Containers	523
Standard C-Style Arrays	523
strings	524
Streams	524
bitset	524
bitset Basics	525
Bitwise Operators	525
bitset Example: Representing Cable Channels	526
Summary	529
CHAPTER 17: MASTERING STL ALGORITHMS	531
Overview of Algorithms	532
The find and find_if Algorithms	532
The accumulate Algorithms	535
Move Semantics with Algorithms	536
Lambda Expressions	536
Syntax	536
Full Syntax	538
Generic Lambda Expressions	539
Lambda Capture Expressions	539
Lambda Expressions as Return Type	540
Lambda Expressions as Parameters	541
Examples with STL Algorithms	541
count_if	542
generate	542
Function Objects	542
Arithmetic Function Objects	543
Transparent Operator Functors	544
Comparison Function Objects	544
Logical Function Objects	545
Bitwise Function Objects	546
Function Object Adapters	546
Binders	546
Negators	548
Calling Member Functions	548
Writing Your Own Function Objects	549
Algorithm Details	550
Iterators	551

Non-Modifying Sequence Algorithms	551
Search Algorithms	551
Comparison Algorithms	553
Utility Algorithms	555
Modifying Sequence Algorithms	556
transform	556
copy	558
move	559
replace	560
remove	561
unique	562
reverse	562
shuffle	562
Operational Algorithms	562
Partition Algorithms	564
Sorting Algorithms	565
Binary Search Algorithms	566
Set Algorithms	566
Minimum/Maximum Algorithms	569
Numerical Processing Algorithms	570
inner_product	570
iota	570
Algorithms Example: Auditing Voter Registrations	570
The Voter Registration Audit Problem Statement	571
The auditVoterRolls Function	571
The getDuplicates Function	572
Testing the auditVoterRolls Function	573
Summary	573
CHAPTER 18: STRING LOCALIZATION AND REGULAR EXPRESSIONS	575
Localization	575
Localizing String Literals	576
Wide Characters	576
Non-Western Character Sets	577
Locales and Facets	579
Using Locales	579
Using Facets	580
Regular Expressions	581
ECMAScript Syntax	582

Anchors	582
Wildcards	582
Alternation	582
Grouping	583
Repetition	583
Precedence	584
Character Set Matches	584
Word Boundaries	586
Back References	587
Lookahead	587
Regular Expressions and Raw String Literals	587
The regex Library	588
regex_match()	589
regex_match() Example	589
regex_search()	592
regex_search() Example	592
regex_iterator	593
regex_iterator Example	593
regex_token_iterator	594
regex_token_iterator Examples	595
regex_replace()	596
regex_replace() Examples	597
Summary	599
CHAPTER 19: ADDITIONAL LIBRARY UTILITIES	601
std::function	601
Ratios	603
The Chrono Library	606
Duration	606
Clock	610
Time Point	611
Random Number Generation	612
Random Number Engines	613
Random Number Engine Adapters	615
Predefined Engines and Engine Adapters	616
Generating Random Numbers	616
Random Number Distributions	618
Tuples	621
Summary	624

PART IV: MASTERING ADVANCED FEATURES OF C++

CHAPTER 20: CUSTOMIZING AND EXTENDING THE STL	627
Allocators	628
Iterator Adapters	628
Reverse Iterators	629
Stream Iterators	630
Insert Iterators	631
Move Iterators	632
Extending the STL	634
Why Extend the STL?	634
Writing an STL Algorithm	634
find_all()	634
Iterator Traits	635
Writing an STL Container	636
A Basic Hash Map	636
Making hash_map an STL Container	644
Note on Allocators	657
Note on Reversible Containers	658
Making hash_map an Associative Container	658
Note on Sequential Containers	670
Summary	671
CHAPTER 21: ADVANCED TEMPLATES	673
More about Template Parameters	673
More about Template Type Parameters	674
Default Values for Template Type Parameters	676
Introducing Template Template Parameters	676
More about Non-Type Template Parameters	678
Reference and Pointer Non-Type Template Parameters	679
Class Template Partial Specialization	679
Another Form of Partial Specialization	681
Emulating Function Partial Specialization with Overloading	683
More on Deduction	684
Template Recursion	685
An N-Dimensional Grid: First Attempt	685
A Real N-Dimensional Grid	686
Type Inference	689
auto and decltype with Templates	689

Variadic Templates	691
Type-Safe Variable-Length Argument Lists	692
Variable Number of Mixin Classes	694
Metaprogramming	695
Factorial at Compile Time	696
Loop Unrolling	696
Printing Tuples	698
Type Traits	699
Using Type Categories	700
Using Type Relations	702
Using <code>enable_if</code>	703
Metaprogramming Conclusion	705
Summary	705
CHAPTER 22: MEMORY MANAGEMENT	707
Working with Dynamic Memory	708
How to Picture Memory	708
Allocation and Deallocation	709
Using <code>new</code> and <code>delete</code>	710
What about My Good Friend <code>malloc</code> ?	710
When Memory Allocation Fails	711
Arrays	711
Arrays of Basic Types	712
Arrays of Objects	713
Deleting Arrays	714
Multi-Dimensional Arrays	715
Working with Pointers	718
A Mental Model for Pointers	718
Casting with Pointers	719
Array-Pointer Duality	720
Arrays Are Pointers!	720
Not All Pointers Are Arrays!	722
Low-Level Memory Operations	722
Pointer Arithmetic	722
Custom Memory Management	723
Garbage Collection	724
Object Pools	725
Function Pointers	725
Pointers to Methods and Data Members	727
Smart Pointers	727
The Old Deprecated <code>auto_ptr</code>	728

The unique_ptr and shared_ptr Smart Pointers	729
unique_ptr	729
shared_ptr	731
Move Semantics	733
weak_ptr	734
Common Memory Pitfalls	734
Underallocating Strings	734
Accessing Out-of-Bounds Memory	735
Memory Leaks	736
Finding and Fixing Memory Leaks in Windows with Visual C++	737
Finding and Fixing Memory Leaks in Linux with Valgrind	738
Double-Deleting and Invalid Pointers	739
Summary	740
CHAPTER 23: MULTITHREADED PROGRAMMING WITH C++	741
Introduction	742
Race Conditions	743
Deadlocks	745
Tearing	746
Cache Coherency	746
Threads	746
Thread with Function Pointer	746
Thread with Function Object	748
Thread with Lambda	750
Thread with Member Function	751
Thread Local Storage	751
Cancelling Threads	752
Retrieving Results from Threads	752
Copying and Rethrowing Exceptions	752
Atomic Operations Library	755
Atomic Type Example	755
Atomic Operations	757
Mutual Exclusion	758
Mutex Classes	759
Non-Timed Mutex Classes	759
Timed Mutex Classes	760
Locks	761
lock_guard	761
unique_lock	761
shared_lock	762
Acquiring Multiple Locks at Once	762

std::call_once	763
Examples Using Mutual Exclusion Objects	764
Thread-Safe Writing to Streams	764
Using Timed Locks	765
Double-Checked Locking	766
Condition Variables	767
Futures	770
Exception Handling	772
Example: Multithreaded Logger Class	772
Thread Pools	776
Threading Design and Best Practices	777
Summary	778

PART V: C++ SOFTWARE ENGINEERING

CHAPTER 24: MAXIMIZING SOFTWARE ENGINEERING METHODS	781
The Need for Process	782
Software Life Cycle Models	783
The Stagewise Model and Waterfall Model	783
Benefits of the Waterfall Model	784
Drawbacks of the Waterfall Model	785
The Spiral Model	785
Benefits of the Spiral Model	786
Drawbacks of the Spiral Model	787
The Rational Unified Process	787
RUP as a Product	788
RUP as a Process	788
RUP in Practice	788
Software Engineering Methodologies	789
Agile	789
Scrum	790
Roles	790
The Process	790
Benefits of Scrum	791
Drawbacks of Scrum	791
Extreme Programming (XP)	792
XP in Theory	792
XP in Practice	795
Software Triage	796
Building Your Own Process and Methodology	796

Be Open to New Ideas	796
Bring New Ideas to the Table	797
Recognize What Works and What Doesn't Work	797
Don't Be a Renegade	797
Source Code Control	797
Summary	799
CHAPTER 25: WRITING EFFICIENT C++	801
Overview of Performance and Efficiency	802
Two Approaches to Efficiency	802
Two Kinds of Programs	802
Is C++ an Inefficient Language?	802
Language-Level Efficiency	803
Handle Objects Efficiently	804
Pass-by-Reference	804
Return-by-Reference	805
Catch Exceptions by Reference	806
Use Move Semantics	806
Avoid Creating Temporary Objects	806
Use Inline Methods and Functions	808
Design-Level Efficiency	808
Cache Where Necessary	808
Cache Invalidation	809
Use Object Pools	809
An Object Pool Implementation	810
Using the Object Pool	812
Profiling	813
Profiling Example with gprof	814
First Design Attempt	814
Profile of the First Attempt	817
Second Attempt	819
Profile of the Second Attempt	821
Profiling Example with Visual C++ 2013	822
Profile of the First Design Attempt	822
Summary	825
CHAPTER 26: CONQUERING DEBUGGING	827
The Fundamental Law of Debugging	828
Bug Taxonomies	828
Avoiding Bugs	828
Planning for Bugs	829

Error Logging	829
Debug Traces	831
Debug Mode	831
Ring Buffers	835
Assertions	839
Static Assertions	840
Crash Dumps	841
Debugging Techniques	842
Reproducing Bugs	842
Debugging Reproducible Bugs	843
Debugging Nonreproducible Bugs	843
Debugging Regressions	844
Debugging Memory Problems	844
Categories of Memory Errors	845
Tips for Debugging Memory Errors	848
Debugging Multithreaded Programs	849
Debugging Example: Article Citations	850
Buggy Implementation of an ArticleCitations Class	850
Testing the ArticleCitations class	852
Lessons from the ArticleCitations Example	861
Summary	861
APPENDIX A: C++ INTERVIEWS	863
Chapter 1: A Crash Course in C++ and the STL	863
Chapters 2 and 18: Strings, Localization, and Regular Expressions	864
Chapter 3: Coding with Style	865
Chapter 4: Designing Professional C++ Programs	866
Chapter 5: Designing with Objects	868
Chapter 6: Designing for Reuse	868
Chapters 7 and 8: Classes and Objects	869
Chapter 9: Discovering Inheritance Techniques	872
Chapter 10: C++ Quirks, Oddities, and Incidentals	873
Chapters 11 and 21: Templates	875
Chapter 12: Demystifying C++ I/O	875
Chapter 13: Handling Errors	876
Chapter 14: Overloading C++ Operators	877
Chapters 15, 16, 17, and 20: The Standard Template Library	878
Chapter 19: Additional Library Utilities	878
Chapter 22: Memory Management	879

Chapter 23: Multithreaded Programming with C++	880
Chapter 24: Maximizing Software Engineering Methods	881
Chapter 25: Writing Efficient C++	882
Chapter 26: Conquering Debugging	882
APPENDIX B: ANNOTATED BIBLIOGRAPHY	885
C++	885
Beginning C++	885
General C++	886
I/O Streams and Strings	887
The C++ Standard Library	887
C++ Templates	888
C++11 / C++14	888
C	889
Unified Modeling Language, Uml	889
Algorithms and Data Structures	890
Random Numbers	890
Open-Source Software	890
Software Engineering Methodology	891
Programming Style	892
Computer Architecture	892
Efficiency	893
Testing	893
Debugging	893
Design Patterns	893
Operating Systems	894
Multithreaded Programming	894
APPENDIX C: STANDARD LIBRARY HEADER FILES	895
The C Standard Library	895
Containers	897
Algorithms, Iterators, and Allocators	898
General Utilities	898
Mathematical Utilities	899
Exceptions	899
I/O Streams	900
Threading Library	901
INDEX	903

INTRODUCTION

FOR MANY YEARS, C++ has served as 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 real-world 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, and the programming techniques that separate novice programmers 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 might not be using the full capabilities of the language. Perhaps you write competent C++ code, but would like to learn more about design in C++ and good programming style. 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 of the language. Chapter 1 covers the basics of C++ as a refresher, but it is not a substitute for actual training and use of the language. 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 hash tables and 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.

You will also need to be familiar with the compiler you will be using to develop your code. This book does not provide detailed directions for using individual compilers. Refer to the documentation that came with your compiler for a refresher.

WHAT THIS BOOK COVERS

Professional C++ is an approach to C++ programming that will both increase the quality of your code and improve your programming efficiency. This third edition of *Professional C++* includes discussions on new C++14 features throughout the book. New C++14 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 more than just the syntax and language features of C++. It also emphasizes programming methodologies 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 can be applied to your actual 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 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 on working with strings because strings are used extensively in examples. 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, “Coding the Professional Way,” provides a technical tour of C++ from the Professional point of view. You will read about how to create reusable classes, and how to leverage important language features such as inheritance. You will also learn about the unusual and quirky parts of the language, techniques for input and output, error handling, string localization, and how to work with regular expressions. You will read about how to implement operator overloading, and how to write templates. This part also explains the C++ Standard Library, including containers, iterators, and algorithms. You will also read about some additional libraries available in the standard such as the libraries to work with time and random numbers.

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, the best ways to manage memory in C++, 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, techniques used to debug C++ programs, and how to write efficient C++ code.

The book concludes with a useful chapter-by-chapter guide to succeeding in a C++ technical interview, an annotated bibliography, and a summary of the C++ header files available in the standard.

This book is not a reference of every single class, method, and function available in C++. Such references are available on the Internet. Two excellent online references are as follows:

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

These online references are continuously updated and extended with sample code and new features, something that is not possible with a book.

This book sometimes refers to such a detailed C++ reference by saying “a Standard Library Reference.”

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.

Note that this book includes new features introduced in the C++14 standard. At the time of this writing, most compilers are not yet fully C++14 compliant.

You can use whichever C++ compiler you like. If you don’t have a C++ compiler yet, you can download a free one. There are a lot of choices. For example, for Windows, you can download Microsoft Visual Studio Express 2013 for Windows Desktop, which is free and includes Visual C++.

For Linux, you can use GCC or Clang, which are also free. Sample code in this book has been tested with Visual C++ and GCC.

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.

Microsoft Visual C++

First, you need to create a project. Start VC++ and click on File \Rightarrow New \Rightarrow Project. In the project template tree on the left, select Visual C++ \Rightarrow Win32. Then select the Win32 Console Application template in the list in the middle of the window. At the bottom specify a name for the project, a location where to save it, and click OK. A wizard opens. Click Next, select Console application and Empty Project, and click Finish.

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, go to View \Rightarrow Solution Explorer. You can add new files or existing files to a project by clicking right on the project name in the Solution Explorer and then Add \Rightarrow New Item or Add \Rightarrow Existing Item.

Use Build \Rightarrow Build Solution to compile your code. When it compiles without errors you can run it with Debug \Rightarrow Start Debugging.

If your program exits before you have a chance to view the output, use Debug \Rightarrow Start without Debugging. It will add a pause to the end of the program so you can view the output.

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:

```
gcc -lstdc++ -std=c++1y -o <executable_name> <source1.cpp> [source2.cpp ...]
```

The -std=c++1y is required to tell GCC to enable C++14 support.

For example, you can compile the `AirlineTicket` example from Chapter 1 by changing to the directory containing the code and running:

```
gcc -lstdc++ -std=c++1y -o AirlineTicket AirlineTicket.cpp AirlineTicketTest.cpp
```

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

```
./AirlineTicket
```

CONVENTIONS

To help you get the most from the text and keep track of what's happening, a number of conventions are used throughout the 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 *highlighted* 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 as: www.wrox.com.

Code is presented in two different ways:

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++14 standard have a little C++14 icon on the left, just as this paragraph does.

SOURCE CODE

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. All of the source code used in this book is available for download at www.wrox.com/go/proc++3e.

Alternatively, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

NOTE *Because many books have similar titles, you may find it easiest to search by ISBN; for this book the ISBN is 978-1-118-85805-9.*

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

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration, and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wrox.com and locate the title by using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors.

A complete book list, including links to each book's errata, is also available at www.wrox.com/mis.../booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies, and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At p2p.wrox.com you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

NOTE *You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.*

Once you've joined, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

PART I

Introduction to Professional C++

- **CHAPTER 1:** A Crash Course in C++ and the STL
- **CHAPTER 2:** Working with Strings
- **CHAPTER 3:** Coding with Style

1

A Crash Course in C++ and the STL

WHAT'S IN THIS CHAPTER?

- A brief overview of the most important parts and syntax of the C++ language and the Standard Template Library (STL)
- The basics of smart pointers

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e 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 base of knowledge before embarking on the rest of the book. This chapter is not a comprehensive lesson in the C++ programming language nor the STL. The basic points, such as what a program is and the difference between = and ==, are not covered. The 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've been away from C++ for a while and you've forgotten the syntax of a `for` loop, you'll find that syntax in this chapter. Also, 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, as well. You'll also learn the basics on how to use the functionality available in the STL, such as `vector` containers, `string` objects, and smart pointers.

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.

THE BASICS OF C++

The C++ language is often viewed as a “better C” or a “superset of C.” Many of the annoyances or rough edges of the C language were addressed when C++ was designed. Because C++ is based on C, much 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 look out for new or unfamiliar syntax!

The Obligatory Hello, World

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

```
// helloworld.cpp
#include <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
- Preprocessor Directives
- The `main()` Function
- I/O Streams

These concepts are briefly explained in the next sections.

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.

Preprocessor Directives

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 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. In C++, declarations usually go into files with extension .h, known as *header files*, while definitions usually go into files with extension .cpp, known as *source files*. A lot of other programming languages do not separate declarations and definitions into separate files, for example C# and Java.

The `<iostream>` header declares the input and output mechanisms provided by C++. If the program did not include it, it would be unable to perform its only task of outputting text.

NOTE In C, header files usually end in .h, such as `<stdio.h>`. In C++, the suffix is omitted for standard library headers, such as `<iostream>`. Standard headers from C still exist in C++, but with new names. For example, you can access the functionality from `<stdio.h>` by including `<cstdio>`.

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.
<code>#define [key] [value]</code>	Every occurrence of the specified key 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 macros. Macros are often dangerous so use them cautiously. See Chapter 10 for details.

continues

(continued)

#ifdef [key] #endif	Code within the <code>#ifdef</code> ("if defined") or <code>#ifndef</code> ("if not defined") blocks are conditionally included or omitted based on whether the specified key has been defined with <code>#define</code> .	Used most frequently to protect against circular includes. Each include file starts with a <code>#ifndef</code> checking the absence of a key, followed by defining that key. The include file ends with a <code>#endif</code> . This prevents the file from being included multiple times; see example after this table.
#pragma [xyz]	<code>xyz</code> varies from compiler to compiler. Often allows the programmer to display a warning or error if the directive is reached during preprocessing.	See example after this table.

One example of using preprocessor directives is to avoid multiple includes. For example:

```
#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, this can be rewritten as follows:

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

Chapter 10 discusses this in more details.

The `main()` Function

`main()` is, of course, where the program starts. An `int` is returned from `main()`, indicating the result status of the program. 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]` might be the program name, but you should not rely on it, and you should never use it.

I/O Streams

I/O streams are covered in depth in Chapter 12, but the basics of output are very 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 data of varying types 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;
```

`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 character*, which refers to a new-line character. Escape characters can be used within any quoted string of text. The following table shows the most common escape characters:

<code>\n</code>	new line
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\\"</code>	the backslash character
<code>\"</code>	quotation mark

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. User input can be tricky because you can never know what kind of data the user will enter. See Chapter 12 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 trusty old `printf()`. While `printf()` can still be used in C++, it's recommended to use the streams library.

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. Suppose the following is in a file called `namespaces.h`:

```
namespace mycode {
    void foo();
}
```

The implementation of a method or function can also be handled in a namespace:

```
#include <iostream>
#include "namespaces.h"
namespace mycode {
```

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

By placing your version of `foo()` in the namespace “`mycode`,” it is isolated 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 the `using` directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the code that follows:

```
#include "namespaces.h"
using namespace mycode;
int main()
{
    foo(); // Implies mycode::foo();
    return 0;
}
```

A single source file can contain multiple `using` directives, but beware of overusing this shortcut. In the extreme case, if you declare that you’re using every namespace known to humanity, you’re effectively eliminating namespaces entirely! Name conflicts will again result if you are using two namespaces that contain the same names. It is also important to know in which namespace your code is operating so that you don’t end up accidentally calling the wrong version of a function.

You’ve seen the namespace syntax before — you used it in the Hello, World program, where `cout` and `endl` are actually names defined in the `std` namespace. You could have written Hello, World with the `using` directive as shown here:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

A `using` declaration can be used to refer to a particular item within a namespace. For example, if the only part of the `std` namespace that you intend to use is `cout`, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to `cout` without prepending the namespace, but other items in the `std` namespace will still need to be explicit:

```
using std::cout;
cout << "Hello, World!" << std::endl;
```

WARNING *Never put a using directive or using declaration in a header file, otherwise you force it on everyone that is including your header.*

Variables

In C++, *variables* can be declared just about anywhere in your code and can be used anywhere in the current block below the line where they are declared. Variables can be declared without being given a value. These uninitialized variables generally end up with a semi-random value based on whatever is in memory at the time and are the source of countless bugs. Variables in C++ can alternatively be assigned an initial value when they are declared. The code that follows shows both flavors of variable declaration, both using `ints`, which represent integer values.

```
int uninitializedInt;
int initializedInt = 7;
cout << uninitializedInt << " is a random value" << endl;
cout << initializedInt << " was assigned an initial value" << endl;
```

NOTE *Most compilers will issue a warning or an error when code is using uninitialized variables. Some compilers will generate code that will report an error at run time.*

The table that follows shows the most common types used in C++.

TYPE	DESCRIPTION	USAGE
<code>int</code> <code>signed</code>	Positive and negative integers; range depends on compiler	<code>int i = -7;</code> <code>signed j = -5;</code>
<code>short (int)</code>	Short integer (usually 2 bytes)	<code>short s = 13;</code>
<code>long (int)</code>	Long integer (usually 4 bytes)	<code>long l = -7L;</code>
<code>long long (int)</code>	Long long integer; range depends on compiler, but at least the same as long (usually 8 bytes)	<code>long long ll = 14LL;</code>
<code>unsigned (int)</code> <code>unsigned short (int)</code> <code>unsigned long (int)</code> <code>unsigned long long (int)</code>	Limits the preceding types to values ≥ 0	<code>unsigned int i = 2U;</code> <code>unsigned j = 5U;</code> <code>unsigned short s = 23U;</code> <code>unsigned long l = 5400UL;</code> <code>unsigned long long ll = 140ULL;</code>

continues

(continued)

float	Floating-point numbers	float f = 7.2f;
double	Double precision numbers; precision is at least the same as for float	double d = 7.2;
long double	Long double precision numbers; precision is at least the same as for double	long double d = 16.98L;
char	A single character	char ch = 'm';
char16_t	A single 16-bit character	char16_t c16 = u'm';
char32_t	A single 32-bit character	char32_t c32 = U'm';
wchar_t	A single wide-character; size depends on compiler	wchar_t w = L'm';
bool	true or false	bool b = true;

NOTE C++ does not provide a basic string type. However, a standard implementation of a string is provided as part of the standard library as described later in this chapter and in more detail in Chapter 2.

Variables can be converted to other types by *casting* them. For example, a `float` can be cast to an `int`. C++ provides three ways of explicitly changing the type of a variable. The first method is a holdover from C, and unfortunately is still commonly used. The second method seems more natural at first but is rarely seen. The third is the most verbose, but the cleanest and recommended method.

```
float myFloat = 3.14f;
int i1 = (int)myFloat;           // method 1
int i2 = int(myFloat);          // method 2
int i3 = static_cast<int>(myFloat); // method 3
```

The resulting integer will be the value of the floating point number with the fractional part truncated. Chapter 10 describes the different casting methods in more detail. In some contexts, variables can be automatically cast, or *coerced*. For example, a `short` can be automatically converted into a `long` because a `long` represents the same type of data with at least the same precision.

```
long someLong = someShort;           // no explicit cast needed
```

When automatically casting variables, you need to be aware of the potential loss of data. For example, casting a `float` to an `int` throws away information (the fractional part of the number). Most compilers will issue a warning if you assign a `float` to an `int` without an explicit cast. If you are certain that the left-hand side type is fully compatible with the right-hand side type, it's okay to cast implicitly.

Literals

Literals are used to write numbers or strings in your code. C++ supports a number of standard literals. Numbers can be specified with the following literals. All four examples represent the same number.

- Decimal literal, for example: 123
- Octal literal, for example: 0173
- Hexadecimal literal, for example: 0x7B
- Binary literal, for example: 0b1111011

Other examples of literals in C++:

- A floating point value: 3.14f
- A double floating point value: 3.14
- A single character: 'a'
- A zero-terminated array of characters: "character array"

It's also possible to define your own type of literals, which is an advanced feature explained in Chapter 10.

 C++14 allows the use of digits separators in numeric literals. A digits separator is a single quote character. For example:

```
int number1 = 23'456'789;      // The number 23456789
float number2 = 0.123'456f;    // The number 0.123456
```

Operators

What good is a variable if you don't have a way to change it? The following table shows the most common *operators* used in C++ and sample code that makes use of them. Note that operators in C++ can be *binary* (operate on two expressions), *unary* (operate on a single expression), or even *ternary* (operate on three expressions). There is only one ternary operator in C++ and it is covered in the section "Conditionals."

OPERATOR	DESCRIPTION	USAGE
=	Binary operator to assign the value on the right to the expression on the left.	int i; i = 3; int j; j = i;
!	Unary operator to complement the true/false (non-0/0) status of an expression.	bool b = !true; bool b2 = !b;

continues

(continued)

+	Binary operator for addition.	int i = 3 + 2; int j = i + 5; int k = i + j;
- * /	Binary operators for subtraction, multiplication, and division.	int i = 5-1; int j = 5*2; int k = j / i;
%	Binary operator for remainder of a division operation. Also referred to as the <i>mod</i> or <i>modulo</i> operator.	int remainder = 5 % 2;
++	Unary operator to increment an expression by 1. If the operator occurs after the expression or <i>post-increment</i> , the result of the expression is the unincremented value. If the operator occurs before the expression or <i>pre-increment</i> , the result of the expression is the new value.	i++; ++i;
--	Unary operator to decrement an expression by 1.	i--; --i;
+=	Shorthand syntax for <code>i = i + j</code>	i += j;
-=	Shorthand syntax for <code>i = i - j;</code>	i -= j;
*=	<code>i = i * j;</code>	i *= j;
/=	<code>i = i / j;</code>	i /= j;
%=	<code>i = i % j;</code>	i %= j;
&	Takes the raw bits of one expression and performs a bitwise "AND" with the other expression.	i = j & k; j &= k;
	Takes the raw bits of one expression and performs a bitwise "OR" with the other expression.	i = j k; j = k;
<< >> <<= >>=	Takes the raw bits of an expression and "shifts" each bit left (<<) or right (>>) the specified number of places.	i = i << 1; i = i >> 4; i <<= 1; i >>= 4;
^ ^=	Performs a bitwise "exclusive or," also called "XOR" operation, on the two expressions.	i = i ^ j; i ^= j;

The following program shows the most common variable types and operators in action. If you're unsure about how variables and operators work, try to figure out what the output of this program will be, and then run it to confirm your answer.

```

int someInteger = 256;
short someShort;
long someLong;
float someFloat;
double someDouble;
someInteger++;
someInteger *= 2;
someShort = static_cast<short>(someInteger);
someLong = someShort * 10000;
someFloat = someLong + 0.785f;
someDouble = static_cast<double>(someFloat) / 100000;
cout << someDouble << endl;

```

The C++ compiler has a recipe for the order in which expressions are evaluated. If you have a complicated line of code with many operators, the order of execution may not be obvious. For that reason, it's probably better to break up a complicated statement into several smaller statements or explicitly group expressions by using parentheses. For example, the following line of code is confusing unless you happen to know the C++ operator precedence table by heart:

```
int i = 34 + 8 * 2 + 21 / 7 % 2;
```

Adding parentheses makes it clear which operations are happening first:

```
int i = 34 + (8 * 2) + (21 / 7) % 2;
```

For those of you playing along at home, both approaches are equivalent and end up with `i` equal to 51. If you assumed that C++ evaluated expressions from left to right, your answer would have been 1. C++ evaluates `/`, `*`, and `%` first (in left-to-right order), followed by addition and subtraction, then bitwise operators. Parentheses let you explicitly tell the compiler that a certain operation should be evaluated separately.

Types

In C++, you can use the basic types (`int`, `bool`, etc.) to build more complex types of your own design. Once you are an experienced C++ programmer, you will rarely use the following techniques, which are features brought in from C, because classes are far more powerful. Still, it is important to know about the following ways of building types so that you will recognize the syntax.

Enumerated Types

An integer really represents a value within a sequence — the sequence of numbers. *Enumerated types* let you define your own sequences so that you can declare variables with values in that sequence. For example, in a chess program, you *could* represent each piece as an `int`, with constants for the piece types, as shown in the following code. The integers representing the types are marked `const` to indicate that they can never change.

```

const int PieceTypeKing = 0;
const int PieceTypeQueen = 1;
const int PieceTypeRook = 2;
const int PieceTypePawn = 3;
//etc.
int myPiece = PieceTypeKing;

```

This representation is fine, but it can become dangerous. Since a piece is just an `int`, what would happen if another programmer added code to increment the value of a piece? By adding one, a king becomes a queen, which really makes no sense. Worse still, someone could come in and give a piece a value of `-1`, which has no corresponding constant.

Enumerated types solve these problems by tightly defining the range of values for a variable. The following code declares a new type, `PieceType`, which has four possible values, representing four of the chess pieces.

```
enum PieceType { PieceTypeKing, PieceTypeQueen, PieceTypeRook, PieceTypePawn };
```

Behind the scenes, an enumerated type is just an integer value. The real value of `PieceTypeKing` is zero. However, by defining the possible values for variables of type `PieceType`, your compiler can give you a warning or error if you attempt to perform arithmetic on `PieceType` variables or treat them as integers. The following code, which declares a `PieceType` variable, and then attempts to use it as an integer, results in a warning or error on most compilers.

```
PieceType myPiece;  
myPiece = 0;
```

It's also possible to specify the integer values for members of an enumeration. The syntax is as follows.

```
enum PieceType { PieceTypeKing = 1, PieceTypeQueen, PieceTypeRook = 10, PieceTypePawn };
```

In this example, `PieceTypeKing` has the integer value 1, `PieceTypeQueen` has the value 2 assigned by the compiler, `PieceTypeRook` has the value 10, and `PieceTypePawn` has the value 11 assigned automatically by the compiler.

The compiler assigns a value to an enumeration member that is the value of the previous enumeration member incremented by one. If you don't assign a value to the first enumeration member yourself, the compiler assigns it the value 0.

Strongly Typed Enumerations

Enumerations as explained above are not strongly typed, meaning they are not type-safe. They are always interpreted as integers, and thus you can compare enumeration values from completely different enumeration types. The `enum class` solves these problems. For example:

```
enum class MyEnum  
{  
    EnumValue1,  
    EnumValue2 = 10,  
    EnumValue3  
};
```

This is a type-safe enumeration called `MyEnum`. These enumeration value names are not automatically exported to the enclosing scope, which means you always have to use the scope resolution operator:

```
MyEnum value1 = MyEnum::EnumValue1;
```

The enumeration values are not automatically converted to integers, which means the following is illegal:

```
if (MyEnum::EnumValue3 == 11) { ... }
```

By default, the underlying type of an enumeration value is an integer, but this can be changed as follows:

```
enum class MyEnumLong : unsigned long
{
    EnumValueLong1,
    EnumValueLong2 = 10,
    EnumValueLong3
};
```

Structs

Structs let you encapsulate one or more existing types into a new type. The classic example of a struct is a database record. If you are building a personnel system to keep track of employee information, you will need to store the first initial, last initial, employee number, and salary for each employee. A struct that contains all of this information is shown in the `employeestruct.h` header file that follows:

```
struct Employee {
    char firstInitial;
    char lastInitial;
    int employeeNumber;
    int salary;
};
```

A variable declared with type `Employee` will have all of these *fields* built-in. The individual fields of a struct can be accessed by using the “.” operator. The example that follows creates and then outputs the record for an employee:

```
#include <iostream>
#include "employeestruct.h"
using namespace std;
int main()
{
    // Create and populate an employee.
    Employee anEmployee;
    anEmployee.firstInitial = 'M';
    anEmployee.lastInitial = 'G';
    anEmployee.employeeNumber = 42;
    anEmployee.salary = 80000;
    // Output the values of an employee.
    cout << "Employee: " << anEmployee.firstInitial <<
        anEmployee.lastInitial << endl;
    cout << "Number: " << anEmployee.employeeNumber << endl;
    cout << "Salary: $" << anEmployee.salary << endl;
    return 0;
}
```

Conditionals

Conditionals let you execute code based on whether or not something is true. As shown in the following sections, there are three main types of conditionals in C++.

if/else Statements

The most common conditional is the `if` statement, which can be accompanied by `else`. If the condition given inside the `if` statement is true, the line or block of code is executed. If not, execution continues to the `else` case if present, or to the code following the conditional. The following pseudocode shows a *cascading if statement*, a fancy way of saying that the `if` statement has an `else` statement that in turn has another `if` statement, and so on.

```
if (i > 4) {  
    // Do something.  
} else if (i > 2) {  
    // Do something else.  
} else {  
    // Do something else.  
}
```

The expression between the parentheses of an `if` statement must be a Boolean value or evaluate to a Boolean value. Logical evaluation operators, described later, provide ways of evaluating expressions to result in a `true` or `false` Boolean value.

switch Statements

The `switch` statement is an alternate syntax for performing actions based on the value of an expression. In C++ `switch` statements, the expression must be of an integral type or of a type convertible to an integral type, and must be compared to constants. Each constant value represents a “case.” If the expression matches the case, the subsequent lines of code are executed until a `break` statement is reached. You can also provide a `default` case, which is matched if none of the other cases match. The following pseudocode shows a common use of the `switch` statement.

```
switch (menuItem) {  
    case OpenMenuItem:  
        // Code to open a file  
        break;  
    case SaveMenuItem:  
        // Code to save a file  
        break;  
    default:  
        // Code to give an error message  
        break;  
}
```

A `switch` statement can always be converted into `if/else` statements. The previous `switch` statement can be converted as follows:

```
if (menuItem == OpenMenuItem) {  
    // Code to open a file  
} else if (menuItem == SaveMenuItem) {  
    // Code to save a file  
} else {  
    // Code to give an error message  
}
```

If a case section omits the `break` statement, the code for that case section is executed first, followed by a *fallthrough*, executing the code for the next case section whether or not that case matches. This

can be a source of bugs, but is sometimes useful. One example is to have a single case section that is executed for several different cases. For example,

```
switch (backgroundColor) {
    case ColorDarkBlue:
    case ColorBlack:
        // Code to execute for both a dark blue or black background color
        break;
    case ColorRed:
        // Code to execute for a red background color
        break;
}
```

`switch` statements are generally used when you want to do something based on the specific value of an expression, as opposed to some test on the expression.

The Conditional Operator

C++ has one operator that takes three arguments, known as a *ternary operator*. It is used as a shorthand conditional expression of the form “if [something] then [perform action], otherwise [perform some other action].” The conditional operator is represented by a `?` and a `:`. The following code will output “yes” if the variable `i` is greater than 2, and “no” otherwise.

```
std::cout << ((i > 2) ? "yes" : "no");
```

The advantage of the conditional operator is that it can occur within almost any context. In the preceding example, the conditional operator is used within code that performs output. A convenient way to remember how the syntax is used is to treat the question mark as though the statement that comes before it really is a question. For example, “Is `i` greater than 2? If so, the result is ‘yes’: if not, the result is ‘no.’”

Unlike an `if` statement or a `switch` statement, the conditional operator doesn’t execute code blocks based on the result. Instead, it is used *within* code, as shown in the preceding example. In this way, it really is an operator (like `+` and `-`) as opposed to a true conditional, such as `if` and `switch`.

Logical Evaluation Operators

You have already seen a *logical evaluation operator* without a formal definition. The `>` operator compares two values. The result is “true” if the value on the left is greater than the value on the right. All logical evaluation operators follow this pattern — they all result in a `true` or `false`.

The following table shows common logical evaluation operators (op):

OP	DESCRIPTION	USAGE
<code><</code>	Determines if the left-hand side is less than, less than or equal to, greater than, or greater than or equal to the right-hand side.	
<code><=</code>		
<code>></code>		
<code>>=</code>		

```
if (i < 0) {
    std::cout << "i is negative";
}
```

continues

(continued)

==	Determines if the left-hand side equals the right-hand side. Don't confuse this with the = (assignment) operator!	<pre>if (i == 3) { std::cout << "i is 3"; }</pre>
!=	Not equals. The result of the statement is true if the left-hand side does <i>not</i> equal the right-hand side.	<pre>if (i != 3) { std::cout << "i is not 3"; }</pre>
!	Logical NOT. Complements the true/false status of a Boolean expression. This is a unary operator.	<pre>if (!someBoolean) { std::cout << "someBoolean is false"; }</pre>
&&	Logical AND. The result is true if both parts of the expression are true.	<pre>if (someBoolean && someOtherBoolean) { std::cout << "both are true"; }</pre>
	Logical OR. The result is true if either part of the expression is true.	<pre>if (someBoolean someOtherBoolean) { std::cout << "at least one is true"; }</pre>

C++ uses *short-circuit logic* when evaluating logical expressions. That means that once the final result is certain, the rest of the expression won't be evaluated. For example, if you are performing a logical OR operation of several Boolean expressions, as shown below, the result is known to be `true` as soon as one of them is found to be `true`. The rest won't even be checked.

```
bool result = bool1 || bool2 || (i > 7) || (27 / 13 % i + 1) < 2;
```

In this example, if `bool1` is found to be `true`, the entire expression must be `true`, so the other parts aren't evaluated. In this way, the language saves your code from doing unnecessary work. It can, however, be a source of hard-to-find bugs if the later expressions in some way influence the state of the program (for example, by calling a separate function). The following code shows a statement using `&&` that will short-circuit after the second term because 0 always evaluates to `false`:

```
bool result = bool1 && 0 && (i > 7) && !done;
```

Arrays

Arrays hold a series of values, all of the same type, each of which can be accessed by its position in the array. In C++, you must provide the size of the array when the array is declared. You cannot give a variable as the size — it must be a constant, or a *constant expression* (`constexpr`), discussed in Chapter 10. The code that follows shows the declaration of an array of three integers followed by three lines to initialize the elements to 0:

```
int myArray[3];
myArray[0] = 0;
```

```
myArray[1] = 0;
myArray[2] = 0;
```

The next section discusses loops which you can use to initialize each element. However, instead of using loops, or using the previous initialization mechanism, you can also accomplish the zero-initialization with the following one-liner:

```
int myArray[3] = {0};
```

Note that this is only possible if you want to initialize all values to zero. For example, the following line fills only the first element in the array with the value 2 and fills all the other elements in the array with the value 0:

```
int myArray[3] = {2};
```

An array can also be initialized with an initializer list, in which case the compiler can deduce the size of the array automatically. For example:

```
int arr[] = {1, 2, 3, 4}; // The compiler creates an array of 4 elements.
```

The preceding examples show a one-dimensional array, which you can think of as a line of integers, each with its own numbered compartment. C++ allows multi-dimensional arrays. You might think of a two-dimensional array as a checkerboard, where each location has a position along the x-axis and a position along the y-axis. Three-dimensional and higher arrays are harder to picture and are rarely used. The following code shows the syntax for allocating a two-dimensional array of characters for a Tic-Tac-Toe board and then putting an “o” in the center square:

```
char ticTacToeBoard[3][3];
ticTacToeBoard[1][1] = 'o';
```

Figure 1-1 shows a visual representation of this board with the position of each square.

TicTacToeBoard[0][0]	TicTacToeBoard[0][1]	TicTacToeBoard[0][2]
TicTacToeBoard[1][0]	TicTacToeBoard[1][1]	TicTacToeBoard[1][2]
TicTacToeBoard[2][0]	TicTacToeBoard[2][1]	TicTacToeBoard[2][2]

FIGURE 1-1

WARNING *In C++, the first element of an array is always at position 0, not position 1! The last position of the array is always the size of the array minus 1!*

std::array

The arrays discussed in the previous section come from C, and still work in C++. However, C++ has a special type of fixed-size container called `std::array`, defined in the `<array>` header file. It's basically a thin wrapper around C-style arrays.

There are a number of advantages in using `std::arrays` instead of C-style arrays. They always know their own size, do not automatically get cast to a pointer to avoid certain types of bugs, and have iterators to easily loop over the elements. Iterators are briefly discussed later in this chapter and are discussed in detail in Chapter 16.

The following example demonstrates how to use the `array` container. The angle brackets after `array`, as in `array<int, 3>`, will become clear during the discussion of templates in Chapter 11. For its use with `array`, it suffices to remember that you have to specify 2 parameters between the angle brackets. The first represents the type of the elements in the array, and the second represents the size of the array.

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
    array<int, 3> arr = {9, 8, 7};
    cout << "Array size = " << arr.size() << endl;
    cout << "Element 2 = " << arr[1] << endl;
    return 0;
}
```

NOTE *Both the C-style arrays and the std::arrays have a fixed size, which should be known at compile time. They cannot grow or shrink at run time.*

If you want an array with a dynamic size, it's recommended to use `std::vector`, explained later in this chapter. A `vector` automatically grows in size when you add new elements to it.

Loops

Computers are great for doing the same thing over and over. C++ provides four looping mechanisms: the `while` loop, `do/while` loop, `for` loop, and *range-based for* loop.

The while Loop

The `while` loop lets you perform a block of code repeatedly as long as an expression evaluates to `true`. For example, the following completely silly code will output "This is silly." five times:

```

int i = 0;
while (i < 5) {
    std::cout << "This is silly." << std::endl;
    ++i;
}

```

The keyword `break` can be used within a loop to immediately get out of the loop and continue execution of the program. The keyword `continue` can be used to return to the top of the loop and reevaluate the `while` expression. Both are often considered poor style because they cause the execution of a program to jump around somewhat haphazardly, so they should be used sparingly. The only place where you have to use `break` is in the context of the `switch` statement, as seen earlier.

The do/while Loop

C++ also has a variation on the `while` loop called `do/while`. It works similarly to the `while` loop, except that the code to be executed comes first, and the conditional check for whether or not to continue happens at the end. In this way, you can use a loop when you want a block of code to always be executed at least once and possibly additional times based on some condition. The example that follows will output “This is silly.” once even though the condition will end up being false:

```

int i = 100;
do {
    std::cout << "This is silly." << std::endl;
    ++i;
} while (i < 5);

```

The for Loop

The `for` loop provides another syntax for looping. Any `for` loop can be converted to a `while` loop and vice versa. However, the `for` loop syntax is often more convenient because it looks at a loop in terms of a starting expression, an ending condition, and a statement to execute at the end of every iteration. In the following code, `i` is initialized to 0; the loop will continue as long as `i` is less than 5; and at the end of every iteration, `i` is incremented by 1. This code does the same thing as the `while` loop example, but is more readable because the starting value, ending condition, and per-iteration statement are all visible on one line.

```

for (int i = 0; i < 5; ++i) {
    std::cout << "This is silly." << std::endl;
}

```

The Range-Based for Loop

The *range-based for* loop is a fourth looping mechanism. It allows for easy iteration over elements of a container. This type of loop works for C-style arrays, initializer lists (discussed in Chapter 10), and any type that has a `begin()` and `end()` method returning iterators, such as `std::array` and all other STL containers discussed in Chapter 16.

The following example first defines an array of four integers. The range-based `for` loop then iterates over a `copy` of every element in this array and prints each value. To iterate over the elements themselves without making copies, use a reference variable, discussed later in this chapter.

```
std::array<int, 4> arr = {1, 2, 3, 4};  
for (int i : arr) {  
    std::cout << i << std::endl;  
}
```

Functions

For programs of any significant size, placing all the code inside of `main()` is unmanageable. To make programs easy to understand, you need to break up, or *decompose*, code into concise functions.

In C++, you first declare a function to make it available for other code to use. If the function is used inside only a particular file, you generally declare and define the function in the source file. If the function is for use by other modules or files, you generally put the declaration in a header file and the definition in a source file.

NOTE *Function declarations are often called “function prototypes” or “signatures” to emphasize that they represent how the function can be accessed, but not the code behind it.*

A function declaration is shown below. This example has a return type of `void`, indicating that the function does not provide a result to the caller. The caller must provide two arguments for the function to work with — an integer and a character.

```
void myFunction(int i, char c);
```

Without an actual definition to match this function declaration, the link stage of the compilation process will fail because code that makes use of the function will be calling nonexistent code. The following definition prints the values of the two parameters:

```
void myFunction(int i, char c)  
{  
    std::cout << "the value of i is " << i << std::endl;  
    std::cout << "the value of c is " << c << std::endl;  
}
```

Elsewhere in the program, you can make calls to `myFunction()` and pass in arguments for the two parameters. Some sample function calls are shown here:

```
myFunction(8, 'a');  
myFunction(someInt, 'b');  
myFunction(5, someChar);
```

NOTE *In C++, unlike C, a function that takes no parameters just has an empty parameter list. It is not necessary to use `void` to indicate that no parameters are taken. However, you must still use `void` to indicate when no value is returned.*

C++ functions can also *return* a value to the caller. The following function adds two numbers and returns the result:

```
int addNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

This function can be called as follows:

```
int sum = addNumbers(5, 3);
```

Every function has a local predefined variable `__func__` that looks as follows:

```
static const char __func__[] = "function-name";
```

This variable can for example be used for logging purposes:

```
int addNumbers(int number1, int number2)
{
    std::cout << "Entering function " << __func__ << std::endl;
    return number1 + number2;
}
```

Alternative Function Syntax

C++ is still using the function syntax as it was designed for C. In the meantime, C++ has been extended with quite a lot of new functionality and this exposed a number of problems with the old function syntax. Since C++11, an alternative function syntax is supported with a *trailing return type*. This new syntax is not of much use for ordinary functions, but is very useful in the context of specifying the return type of template functions. Templates are studied in detail in Chapters 11 and 21.

The following example demonstrates the alternative function syntax. The `auto` keyword in this context has the meaning of starting a function prototype using the alternative function syntax.

```
auto func(int i) -> int
{
    return i + 2;
}
```

The return type of the function is no longer at the beginning, but placed at the end of the line after the arrow, `->`. The following code demonstrates that calling `func()` remains exactly the same and shows that the `main()` function can also use this alternative syntax:

```
auto main() -> int
{
    cout << func(3) << endl;
    return 0;
}
```

Function Return Type Deduction

C++14 allows you to ask the compiler to figure out the return type of a function automatically. To make use of this functionality, you need to specify `auto` as the return type and omit any trailing return type:

```
auto divideNumbers(double numerator, double denominator)
{
    if (denominator == 0) { /* ... */ }
    return numerator / denominator;
}
```



The compiler deduces the return type based on the expressions used for the `return` statements. There can be multiple `return` statements in the function, but they should all resolve to the same type. Such a function can even include recursive calls (calls to itself), but the first `return` statement in the function must be a non-recursive call.

Type Inference Part 1

Type inference allows the compiler to automatically deduce the type of an expression. There are two keywords for type inference: `auto` and `decltype`, which are discussed in the next two sections.

C++14 adds `decltype(auto)` to the mix, discussed in the section “Type Inference Part 2” later in the chapter because you need to learn about references first before you can understand `decltype(auto)`.

The `auto` Keyword

The `auto` keyword has four completely different meanings. The first meaning is to tell the compiler to automatically deduce the type of a variable at compile time. The following line shows the simplest use of the `auto` keyword in that context:

```
auto x = 123;      // x will be of type int
```

In this example you don’t win much by typing `auto` instead of `int`; however, it becomes useful for more complicated types. Suppose you have a function called `getFoo()` that has a complicated return type. If you want to assign the result of the call to a variable, you can spell out the complicated type, or you can simply use `auto` and let the compiler figure it out:

```
auto result = getFoo();
```

More concrete examples of the `auto` keyword will pop up throughout the book.

The second use of the `auto` keyword is for the *alternative function syntax*, explained earlier.

 The third use of the `auto` keyword is for *function return type deduction*, also discussed earlier. Lastly, a fourth use of `auto` is for generic lambda expressions, discussed in Chapter 17.

The `decltype` Keyword

The `decltype` keyword takes an expression as argument, and computes the type of that expression. For example:

```
int x = 123;
decltype(x) y = 456;
```

In this example, the compiler deduces the type of `y` to be `int` because that’s the type of `x`. Like the `auto` keyword for the alternative function syntax, the `decltype` keyword doesn’t seem to add much value on first sight. However, in the context of templates, discussed in Chapter 11 and 21, `auto` and `decltype` become pretty powerful.

Those Are the Basics

At this point, you have reviewed the basic essentials of C++ programming. If this section was a breeze, skim the next section to make sure that you’re up to speed on the more-advanced material.

If you struggled with this section, you may want to obtain one of the fine introductory C++ books mentioned in Appendix B before continuing.

DIVING DEEPER INTO C++

Loops, variables, and conditionals are terrific building blocks, but there is much more to learn. The topics covered next include many features designed to help C++ programmers with their code as well as a few features that are often more confusing than helpful. If you are a C programmer with little C++ experience, you should read this section carefully.

Pointers and Dynamic Memory

Dynamic memory allows you to build programs with data that is not of fixed size at compile time. Most nontrivial programs make use of dynamic memory in some form.

The Stack and the Heap

Memory in your C++ application is divided into two parts — the *stack* and the *heap*. One way to visualize the stack is as a deck of cards. The current top card represents the current scope of the program, usually the function that is currently being executed. All variables declared inside the current function will take up memory in the top stack frame, the top card of the deck. If the current function, which I'll call `foo()` calls another function `bar()`, a new card is put on the deck so that `bar()` has its own *stack frame* to work with. Any parameters passed from `foo()` to `bar()` are copied from the `foo()` stack frame into the `bar()` stack frame. Figure 1-2 shows what the stack might look like during the execution of a hypothetical function `foo()` that has declared two integer values.

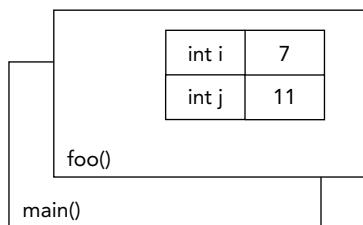


FIGURE 1-2

Stack frames are nice because they provide an isolated memory workspace for each function. If a variable is declared inside the `foo()` stack frame, calling the `bar()` function won't change it unless you specifically tell it to. Also, when the `foo()` function is done running, the stack frame goes away, and all of the variables declared within the function no longer take up memory. Variables that are stack-allocated do not need to be deallocated (deleted) by the programmer; it happens automatically.

The *heap* is an area of memory that is completely independent of the current function or stack frame. You can put variables on the heap if you want them to exist even when the function in which they were created has completed. The heap is less structured than the stack. You can think

of it as just a pile of bits. Your program can add new bits to the pile at any time or modify bits that are already in the pile. You have to make sure that you deallocate (delete) any memory that you allocated on the heap. This does not happen automatically, unless you use smart pointers, discussed in a next section.

Working with Pointers

You can put anything on the heap by explicitly allocating memory for it. For example, to put an integer on the heap, you need to allocate memory for it, but first you need to declare a *pointer*:

```
int* myIntegerPointer;
```

The * after the int type indicates that the variable you are declaring refers/points to some integer memory. Think of the pointer as an arrow that points at the dynamically allocated heap memory. It does not yet point to anything specific because you haven't assigned it to anything; it is an uninitialized variable. Uninitialized variables should be avoided at all times, and especially uninitialized pointers because they point to some random place in memory. Working with such pointers most likely will make your program crash. That's why you should always declare and initialize your pointers at the same time. You can initialize them to a null pointer (nullptr) if you don't want to allocate memory right away:

```
int* myIntegerPointer = nullptr;
```

You use the new operator to allocate the memory:

```
myIntegerPointer = new int;
```

In this case, the pointer points to the address of just a single integer value. To access this value, you need to *dereference* the pointer. Think of dereferencing as following the pointer's arrow to the actual value in the heap. To set the value of the newly allocated heap integer, you would use code like the following:

```
*myIntegerPointer = 8;
```

Notice that this is not the same as setting myIntegerPointer to the value 8. You are not changing the pointer; you are changing the memory that it points to. If you were to reassign the pointer value, it would point to the memory address 8, which is probably random garbage that will eventually make your program crash.

After you are finished with your dynamically allocated memory, you need to deallocate the memory using the delete operator. To prevent the pointer from being used after having deallocated the memory it points to, it's recommended to set your pointer to nullptr:

```
delete myIntegerPointer;  
myIntegerPointer = nullptr;
```

WARNING A pointer must be valid before dereferencing it. A null or uninitialized pointer will cause a crash if dereferenced.

Pointers don't always point to heap memory. You can declare a pointer that points to a variable on the stack, even another pointer. To get a pointer to a variable, you use the & “address of” operator:

```
int i = 8;
int* myIntegerPointer = &i; // Points to the variable with the value 8
```

C++ has a special syntax for dealing with pointers to structures. Technically, if you have a pointer to a structure, you can access its fields by first dereferencing it with `*`, then using the normal `.` syntax, as in the code that follows, which assumes the existence of a function called `getEmployee()`.

```
Employee* anEmployee = getEmployee();
cout << (*anEmployee).salary << endl;
```

This syntax is a little messy. The `->` (arrow) operator lets you perform both the dereference and the field access in one step. The following code is equivalent to the preceding code, but is easier to read:

```
Employee* anEmployee = getEmployee();
cout << anEmployee->salary << endl;
```

Normally, when you pass a variable into a function, you are *passing by value*. If a function takes an integer parameter, it is really a copy of the integer that you pass in. Pointers to stack variables are often used in C to allow functions to modify variables in other stack frames, essentially *passing by reference*. By dereferencing the pointer, the function can change the memory that represents the variable even though that variable isn't in the current stack frame. This is less common in C++, because C++ has a better mechanism, called *references*, which is covered later in this chapter.

Dynamically Allocated Arrays

The heap can also be used to dynamically allocate arrays. You use the `new[]` operator to allocate memory for an array:

```
int arraySize = 8;
int* myVariableSizedArray = new int[arraySize];
```

This allocates memory for enough integers to satisfy the `arraySize` variable. Figure 1-3 shows what the stack and the heap both look like after this code is executed. As you can see, the pointer variable still resides on the stack, but the array that was dynamically created lives on the heap.

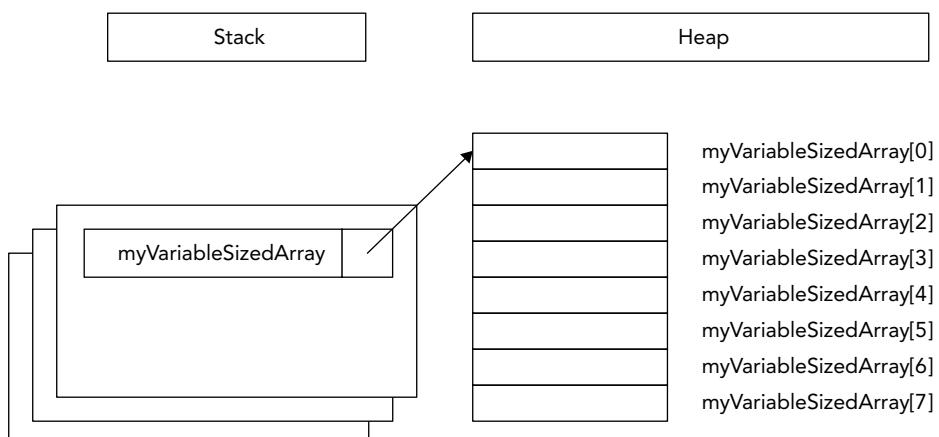


FIGURE 1-3

Now that the memory has been allocated, you can work with `myVariableSizedArray` as though it were a regular stack-based array:

```
myVariableSizedArray[3] = 2;
```

When your code is done with the array, it should remove it from the heap so that other variables can use the memory. In C++, you use the `delete[]` operator to do this.

```
delete[] myVariableSizedArray;
```

The brackets after `delete` indicate that you are deleting an array!

NOTE *Avoid using `malloc()` and `free()` from C. Instead, use `new` and `delete`, or `new[]` and `delete[]`.*

WARNING *To prevent memory leaks, every call to `new` should be paired with a call to `delete`, and every call to `new[]` should be paired with a call to `delete[]`. Pairing a `new[]` with a `delete` also causes a memory leak. Memory leaks are discussed in Chapter 22.*

Null Pointer Constant

Before C++11, the constant 0 was used to define either the number 0 or a null pointer. This can cause some problems. Take the following example:

```
void func(char* str) {cout << "char* version" << endl;}
void func(int i) {cout << "int version" << endl;}
int main()
{
    func(NULL);
    return 0;
}
```

The `main()` function is calling `func()` with parameter `NULL`, which is supposed to be a null pointer constant. In other words, you are expecting the `char*` version of `func()` to be called with a null pointer as argument. However, since `NULL` is not a pointer, but identical to the integer 0, the integer version of `func()` is called.

This problem is solved with the introduction of a real null pointer constant, `nullptr`. The following code calls the `char*` version.

```
func(nullptr);
```

Smart Pointers

To avoid common memory problems, you should use *smart pointers* instead of normal “naked” C-style pointers. Smart pointers automatically deallocate memory when the smart pointer object goes out of scope, for example when the function has finished executing.

There are three smart pointer types in C++: `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr`, all defined in the `<memory>` header. The `unique_ptr` is analogous to an ordinary pointer, except

that it will automatically free the memory or resource when the `unique_ptr` goes out of scope or is deleted. A `unique_ptr` has sole ownership of the object pointed to. One advantage of the `unique_ptr` is that it simplifies coding where storage must be freed when an exceptional situation occurs. When the smart pointer variable leaves its scope, the storage is automatically freed. You can also store a C-style array in a `unique_ptr`. Use `std::make_unique<>()` to create a `unique_ptr`.

For example, instead of writing the following:

```
Employee* anEmployee = new Employee;
```

You should write:

```
auto anEmployee = std::make_unique<Employee>();
```

`make_unique()` is available since C++14. If your compiler is not yet C++14 compliant you can make your `unique_ptr` as follows. Note that you now have to specify the type, `Employee`, twice:

```
std::unique_ptr<Employee> anEmployee(new Employee);
```

You can use the `anEmployee` smart pointer the same way as a normal pointer.

`unique_ptr` is a generic smart pointer that can point to any kind of memory. That's why it is a template. Templates require the angle brackets to specify the template parameters. Between the brackets you have to specify the type of memory you want your `unique_ptr` to point to. Templates are discussed in detail in Chapters 11 and 21, but the smart pointers are introduced in the beginning of the book so that they can be used throughout the book, and as you can see, they are easy to use.

`shared_ptr` allows for distributed “ownership” of the data. Each time a `shared_ptr` is assigned, a reference count is incremented indicating there is one more “owner” of the data. When a `shared_ptr` goes out of scope, the reference count is decremented. When the reference count goes to zero it means there is no longer any owner of the data, and the object referenced by the pointer is freed. You cannot store an array in a `shared_ptr`. Use `std::make_shared<>()` to create a `shared_ptr`.

You can use `weak_ptr` to observe a `shared_ptr` without incrementing or decrementing the reference count of the linked `shared_ptr`.

Chapter 22 discusses these smart pointers in detail, but because the basic use is straightforward, they are already used throughout examples in the book.

NOTE *Naked, plain old pointers are only allowed if there is no ownership involved. Otherwise, use `unique_ptr` by default, and `shared_ptr` if you need shared ownership. If you know about `auto_ptr`, forget it because the C++ standard has deprecated it.*

References

The pattern for most functions is that they take in zero or more parameters, do some calculations, and return a single result. Sometimes, however, that pattern is broken. You may be tempted to return two values, or you may want the function to be able to change the value of one of the variables that were passed in.

In C, the primary way to accomplish such behavior is to pass in a pointer to the variable instead of the variable itself. The only problem with this approach is that it brings the messiness of pointer syntax into what is really a simple task. In C++, there is an explicit mechanism for “pass-by-reference.” Attaching & to a type indicates that the variable is a reference. It is still used as though it was a normal variable, but behind the scenes, it is really a pointer to the original variable. Below are two implementations of an `addOne()` function. The first will have no effect on the variable that is passed in because it is passed by value. The second uses a reference and thus changes the original variable.

```
void addOne(int i)
{
    i++; // Has no real effect because this is a copy of the original
}
void addOne(int& i)
{
    i++; // Actually changes the original variable
}
```

The syntax for the call to the `addOne()` function with an integer reference is no different than if the function just took an integer.

```
int myInt = 7;
addOne(myInt);
```

NOTE *There is a subtle difference between the two `addOne()` implementations. The version using pass-by-value will accept constants without a problem; for example “`addOne(3)` ;” is legal. However, doing the same with the pass-by-reference version of `addOne()` will result in a compiler error. This can be solved by using rvalue references, which is an advanced C++ feature explained in Chapter 10.*

Strings in C++

There are three ways to work with strings of text in C++: the C-style, which represents strings as arrays of characters; the C++ style, which wraps that representation in an easier-to-use string type; and the general class of nonstandard approaches. Chapter 2 provides a detailed discussion.

For now, the only thing you need to know is that the C++ string type is defined in the `<string>` header file, and that you can use a C++ string almost like a basic type. Just like I/O streams, the `string` type lives in the `std` namespace. The example that follows shows how strings can be used just like character arrays.

```
std::string myString = "Hello, World";
cout << "The value of myString is " << myString << endl;
```

Exceptions

C++ is a very flexible language, but not a particularly safe one. The compiler will let you write code that scribbles on random memory addresses or tries to divide by zero (computers don’t deal well with infinity). One of the language features that attempts to add a degree of safety back to the language is *exceptions*.

An exception is an unexpected situation. For example, if you are writing a function that retrieves a web page, several things could go wrong. The Internet host that contains the page might be down, the page might come back blank, or the connection could be lost. One way you could handle this situation is by returning a special value from the function, such as `nullptr` or an error code. Exceptions provide a much better mechanism for dealing with problems.

Exceptions come with some new terminology. When a piece of code detects an exceptional situation, it *throws* an exception. Another piece of code *catches* the exception and takes appropriate action. The following example shows a function, `divideNumbers()`, that throws an exception if the caller passes in a denominator of zero:

```
#include <stdexcept>
double divideNumbers(double numerator, double denominator)
{
    if (denominator == 0) {
        throw std::invalid_argument("Denominator cannot be 0.");
    }
    return numerator / denominator;
}
```

When the `throw` line is executed, the function immediately ends without returning a value. If the caller surrounds the function call with a `try/catch` block, as shown in the following code, it receives the exception and is able to handle it.

```
#include <iostream>
#include <exception>
int main()
{
    try {
        std::cout << divideNumbers(2.5, 0.5) << std::endl;
        std::cout << divideNumbers(2.3, 0) << std::endl;
        std::cout << divideNumbers(4.5, 2.5) << std::endl;
    } catch (const std::exception& exception) {
        std::cout << "Exception caught: " << exception.what() << std::endl;
    }
    return 0;
}
```

The first call to `divideNumbers()` executes successfully, and the result is output to the user. The second call throws an exception. No value is returned, and the only output is the error message that is printed when the exception is caught. The third call is never executed because the second call throws an exception causing the program to jump to the `catch` block. The output for the preceding block of code is:

```
5
An exception was caught: Denominator cannot be 0.
```

Exceptions can get tricky in C++. To use exceptions properly, you need to understand what happens to the stack variables when an exception is thrown, and you have to be careful to properly catch and handle the necessary exceptions. The preceding example used the built-in `std::invalid_argument` type, but it is preferable to write your own exception types that are more specific to the error being thrown. Unlike the Java language, the C++ compiler doesn't force you to catch every exception that might occur. If your code never catches any exceptions but an exception is thrown, it will be caught

by the program itself, which will be terminated. These trickier aspects of exceptions are covered in much more detail in Chapter 13.

The Many Uses of `const`

The keyword `const` can be used in several different ways in C++. All of its uses are related, but there are subtle differences. The subtleties of `const` make for excellent interview questions! Chapter 10 explains in detail all the ways that `const` can be used. The following sections outline the most frequent uses.

`const` Constants

If you assumed that the keyword `const` has something to do with constants, you have correctly uncovered one of its uses. In the C language, programmers often use the preprocessor `#define` mechanism to declare symbolic names for values that won't change during the execution of the program, such as the version number. In C++, programmers are encouraged to avoid `#define` in favor of using `const` to define constants. Defining a constant with `const` is just like defining a variable, except that the compiler guarantees that code cannot change the value.

```
const float versionNumber = 2.0f;
const std::string productName = "Super Hyper Net Modulator";
```

`const` to Protect Parameters

In C++, you can cast a non-`const` variable to a `const` variable. Why would you want to do this? It offers some degree of protection from other code changing the variable. If you are calling a function that a coworker of yours is writing, and you want to ensure that the function doesn't change the value of a parameter you pass in, you can tell your coworker to have the function take a `const` parameter. If the function attempts to change the value of the parameter, it will not compile.

In the following code, a `string*` is automatically cast to a `const string*` in the call to `mysteryFunction()`. If the author of `mysteryFunction()` attempts to change the value of the passed string, the code will not compile. There are ways around this restriction, but using them requires conscious effort. C++ only protects against accidentally changing `const` variables.

```
void mysteryFunction(const std::string* someString)
{
    *someString = "Test"; // Will not compile.
}

int main()
{
    std::string myString = "The string";
    mysteryFunction(&myString);
    return 0;
}
```

`const` References

You will often find code that uses `const` reference parameters. At first, that seems like a contradiction. Reference parameters allow you to change the value of a variable from within another context. `const` seems to prevent such changes.

The main value in `const` reference parameters is efficiency. When you pass a value into a function, an entire copy is made. When you pass a reference, you are really just passing a pointer to the original so the computer doesn't need to make the copy. By passing a `const` reference, you get the best of both worlds — no copy is made but the original variable cannot be changed.

`const` references become more important when you are dealing with objects because they can be large and making copies of them can have unwanted side effects. Subtle issues like this are covered in Chapter 10. The following example shows how to pass a `std::string` to a function as a `const` reference:

```
void printString(const std::string& myString)
{
    std::cout << myString << std::endl;
}
int main()
{
    std::string someString = "Hello World";
    printString(someString);
    return 0;
}
```

Type Inference Part 2

Now that you know about references, `const`, and `std::string`, it's time to revisit type inference with a discussion of `decltype(auto)`.

decltype(auto)

C++14

Using `auto` to deduce the type of an expression strips away reference qualifiers and `const` qualifiers. `decltype` does not strip those but might cause code duplication. C++14 solves this by introducing `decltype(auto)`.

For example, suppose you have the following function:

```
const string message = "Test";

const string& foo()
{
    return message;
}
```

You can call `foo()` and store the result in a variable with the type specified as `auto` as follows:

```
auto f1 = foo();
```

Because `auto` strips reference and `const` qualifiers, `f1` is of type `string`, and thus a copy is made. If you want `f1` to be a `const` reference, you can explicitly make it a reference and mark it `const` as follows:

```
const auto& f1 = foo();
```

An alternative solution is to use `decltype`, which does not strip anything:

```
decltype(foo()) f2 = foo();
```

In this case, `f2` is of type `const string&`; however, there is code duplication because you need to specify `foo()` twice, which can be cumbersome when `foo()` is a more complicated expression.

The solution in C++14 is as follows:

```
decltype(auto) f3 = foo();  
f3 is also of type const string&.
```

C++ AS AN OBJECT-ORIENTED LANGUAGE

If you are a C programmer, you may have viewed the features covered so far in this chapter as convenient additions to the C language. As the name C++ implies, in many ways the language is just a “better C.” There is one major point that this view overlooks. Unlike C, C++ is an object-oriented language.

Object-oriented programming (OOP) is a very different, arguably more natural, way to write code. If you are used to procedural languages such as C or Pascal, don’t worry. Chapter 5 covers all the background information you need to know to shift your mindset to the object-oriented paradigm. If you already know the theory of OOP, the rest of this section will get you up to speed (or refresh your memory) on basic C++ object syntax.

Defining a Class

A *class* defines the characteristics of an object. In C++, classes are usually defined in a header file (.h), while the definitions of its non-inline methods and of any static data members is in a corresponding source file (.cpp).

A basic class definition for an airline ticket class is shown below. The class can calculate the price of the ticket based on the number of miles in the flight and whether or not the customer is a member of the “Elite Super Rewards Program.” The definition begins by declaring the class name. Inside a set of curly braces, the *data members* (properties) of the class and its *methods* (behaviors) are declared. Each data member and method is associated with a particular access level: `public`, `protected`, or `private`. These labels can occur in any order and can be repeated. Members that are `public` can be accessed from outside the class, while members that are `private` cannot be accessed from outside the class. It’s recommended to make all your data members `private`, and if needed, give access to them with `public` getters and setters. This way you can easily change the representation of your data while keeping the `public` interface the same.

```
#include <string>  
class AirlineTicket  
{  
public:  
    AirlineTicket();  
    ~AirlineTicket();  
    int calculatePriceInDollars() const;  
    const std::string& getPassengerName() const;  
    void setPassengerName(const std::string& name);  
    int getNumberOfMiles() const;  
    void setNumberOfMiles(int miles);  
    bool getHasEliteSuperRewardsStatus() const;  
    void setHasEliteSuperRewardsStatus(bool status);  
private:  
    std::string mPassengerName;
```

```

        int mNumberOfMiles;
        bool mHasEliteSuperRewardsStatus;
    };

```

The method that has the same name as the class with no return type is a *constructor*. It is automatically called when an object of the class is created. The method with a tilde (~) character followed by the class name is a *destructor*. It is automatically called when the object is destroyed.

NOTE *To follow the const-correctness principle, it's always a good idea to declare member functions that do not change any data member of the object as being const. These member functions are also called "inspectors," compared to "mutators" for non-const member functions.*

The definitions of some of the `AirlineTicket` class methods are shown below.

```

AirlineTicket::AirlineTicket()
{
    // Initialize data members
    mHasEliteSuperRewardsStatus = false;
    mPassengerName = "Unknown Passenger";
    mNumberOfMiles = 0;
}
AirlineTicket::~AirlineTicket()
{
    // Nothing much to do in terms of cleanup
}
int AirlineTicket::calculatePriceInDollars() const
{
    if (getHasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }
    // The cost of the ticket is the number of miles times
    // 0.1. Real airlines probably have a more complicated formula!
    return static_cast<int>(getNumberOfMiles() * 0.1);
}
const string& AirlineTicket::getPassengerName() const
{
    return mPassengerName;
}
void AirlineTicket::setPassengerName(const string& name)
{
    mPassengerName = name;
}
// Other get and set methods omitted for brevity.

```

The sample program that follows makes use of the `AirlineTicket` class. This example shows the creation of a stack-based `AirlineTicket` object as well as a heap-based object.

```

AirlineTicket myTicket; // Stack-based AirlineTicket
myTicket.setPassengerName("Sherman T. Socketwrench");
myTicket.setNumberOfMiles(700);
int cost = myTicket.calculatePriceInDollars();

```

```
cout << "This ticket will cost $" << cost << endl;

// Heap-based AirlineTicket with smart pointer
auto myTicket2 = make_unique<AirlineTicket>();
myTicket2->setPassengerName("Laudimore M. Hallidue");
myTicket2->setNumberOfMiles(2000);
myTicket2->setHasEliteSuperRewardsStatus(true);
int cost2 = myTicket2->calculatePriceInDollars();
cout << "This other ticket will cost $" << cost2 << endl;
// No need to delete myTicket2, happens automatically

// Heap-based AirlineTicket without smart pointer (not recommended)
AirlineTicket* myTicket3 = new AirlineTicket();
// ... Use ticket 3
delete myTicket3; // delete the heap object!
```

The preceding example exposes you to the general syntax for creating and using classes. Of course, there is much more to learn. Chapters 7, 8, and 9 go into more depth about the specific C++ mechanisms for defining classes.

THE STANDARD LIBRARY

C++ comes with a standard library, which contains a lot of useful classes that can easily be used in your code. The benefit of using classes from the standard library is that you don't need to reinvent certain classes and you don't need to waste time on implementing things that have already been implemented for you. Another benefit is that the classes available in the standard library are heavily tested and verified for correctness by thousands of users. The standard library classes are also tuned for high performance, so using them will most likely result in better performance compared to making your own implementation.

The amount of functionality available to you in the standard library is pretty big. Chapter 15 and later chapters provide more details about the standard library. When you start working with C++ it is a good idea to understand what the standard library can do for you from the very beginning. This is especially important if you are a C programmer. As a C programmer, you might try to solve problems in C++ the same way you would solve them in C. However, in C++ there is probably an easier and safer solution to the problem by using standard library classes.

You already saw some standard library classes earlier in this chapter; for example, `std::string`, `std::array`, and the `std::unique_ptr` smart pointer.

std::vector

Another example of functionality provided by the standard library is the concept of containers, which is used further in this chapter. Take `std::vector` as an example, declared in `<vector>`. The `vector` replaces the concept of C arrays with a much more flexible and safer mechanism. As a user, you need not worry about memory management, as the `vector` will automatically allocate enough memory to hold its elements. A `vector` is dynamic, meaning that elements can be added and removed at run time. To make it easy to loop over the contents of containers, the standard library provides a concept called *iterators*. Chapter 16 goes into more detail regarding containers and

iterators, but the basic use is straightforward, which is why it's introduced in the beginning of the book so that it can be used in examples. The following example demonstrates the basic functionality of the `std::vector` class and the concept of iterators.

```
#include <string>
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;
int main()
{
    // Create a vector of strings, using uniform initialization
    vector<string> myVector = {"A first string", "A second string"};
    // Add some strings to the vector using push_back
    myVector.push_back("A third string");
    myVector.push_back("The last string in the vector");
    // Print the elements using a range-based for loop
    for (const auto& str : myVector)
        cout << str << endl;
    // Iterate over the elements in the vector and print them once more
    for (auto iterator = cbegin(myVector);
         iterator != cend(myVector); ++iterator) {
        cout << *iterator << endl;
    }
    return 0;
}
```

`myVector` is declared as `vector<string>`. The angle brackets are required to specify the template parameters, just as with `std::unique_ptr` earlier in this chapter. A `vector` is a generic container. It can contain almost any kind of object; that's why you have to specify the type of object you want in your `vector` between those brackets. Templates are discussed in detail in Chapters 11 and 21.

To add elements to the `vector`, you can use uniform initialization, discussed in detail in Chapter 10, or the `push_back()` method. Because the type of the `iterator` variable is `auto` and because `cbegin()` is used to initialize the iterator, the compiler automatically deduces its type as `vector<string>::const_iterator`.

YOUR FIRST USEFUL C++ PROGRAM

The following program builds on the employee database example used earlier when discussing structs. This time, you will end up with a fully functional C++ program that uses many of the features discussed in this chapter. This real-world example includes the use of classes, exceptions, streams, vectors, iterators, namespaces, references, and other language features.

An Employee Records System

A program to manage a company's employee records needs to be flexible and have useful features. The feature set for this program includes the following:

- The ability to add an employee
- The ability to fire an employee

- The ability to promote an employee
- The ability to view all employees, past and present
- The ability to view all current employees
- The ability to view all former employees

The design for this program divides the code into three parts. The `Employee` class encapsulates the information describing a single employee. The `Database` class manages all the employees of the company. A separate `UserInterface` file provides the interactivity of the program.

The Employee Class

The `Employee` class maintains all the information about an employee. Its methods provide a way to query and change that information. `Employees` also know how to display themselves on the console. Methods also exist to adjust the employee's salary and employment status.

`Employee.h`

The `Employee.h` file defines the `Employee` class. The sections of this file are described individually in the material that follows.

The first line contains a `#pragma once` to prevent the file from being included multiple times, followed by the inclusion of the `string` functionality.

This code also declares that the subsequent code, contained within the curly braces, lives in the `Records` namespace. `Records` is the namespace that is used throughout this program for application-specific code.

```
#pragma once
#include <string>
namespace Records {
```

The following constant, representing the default starting salary for new employees, lives in the `Records` namespace. Other code that lives in `Records` can access this constant as `kDefaultStartingSalary`. Elsewhere, it must be referenced as `Records::kDefaultStartingSalary`.

```
const int kDefaultStartingSalary = 30000;
```

The `Employee` class is defined, along with its public methods. The `promote()` and `demote()` methods both have integer parameters that are specified with a default value. In this way, other code can omit the integer parameters and the default will automatically be used.

A number of setters and getters provide mechanisms to change the information about an employee or to query the current information about an employee.

```
class Employee
{
public:
    Employee();
    void promote(int raiseAmount = 1000);
    void demote(int demeritAmount = 1000);
    void hire(); // Hires or rehires the employee
```

```

        void fire(); // Dismisses the employee
        void display() const; // Outputs employee info to console

        // Getters and setters
        void setFirstName(const std::string& firstName);
        const std::string& getFirstName() const;

        void setLastName(const std::string& lastName);
        const std::string& getLastName() const;

        void setEmployeeNumber(int employeeNumber);
        int getEmployeeNumber() const;

        void setSalary(int newSalary);
        int getSalary() const;

        bool getIsHired() const;
    
```

Finally, the data members are declared as `private` so that other parts of the code cannot modify them directly. The setters and getters provide the only public way of modifying or querying these values.

```

private:
    std::string mFirstName;
    std::string mLastName;
    int mEmployeeNumber;
    int mSalary;
    bool mHired;
};

}
    
```

Employee.cpp

This section discusses the implementations for the `Employee` member functions. The `Employee` constructor sets the initial values for the `Employee`'s data members. By default, new employees have no name, an employee number of `-1`, the default starting salary, and a status of not hired. This constructor implementation shows a second mechanism to initialize class member variables. You can either put the initialization between the curly braces in the body of the constructor, or you can use a constructor initializer, which follows a colon after the constructor name.

```

#include <iostream>
#include "Employee.h"
using namespace std;
namespace Records {
    Employee::Employee()
        : mFirstName("")
        , mLastName("")
        , mEmployeeNumber(-1)
        , mSalary(kDefaultStartingSalary)
        , mHired(false)
    {
    }
}
    
```

The `promote()` and `demote()` methods simply call the `setSalary()` method with a new value. Note that the default values for the integer parameters do not appear in the source file; they are only allowed in a function declaration, not in a definition.

```
void Employee::promote(int raiseAmount)
{
    setSalary(getSalary() + raiseAmount);
}
void Employee::demote(int demeritAmount)
{
    setSalary(getSalary() - demeritAmount);
}
```

The `hire()` and `fire()` methods just set the `mHired` data member appropriately.

```
void Employee::hire()
{
    mHired = true;
}
void Employee::fire()
{
    mHired = false;
}
```

The `display()` method uses the console output stream to display information about the current employee. Because this code is part of the `Employee` class, it *could* access data members, such as `mSalary`, directly instead of using `getSalary()`. However, it is considered good style to make use of getters and setters when they exist, even within the class.

```
void Employee::display() const
{
    cout << "Employee: " << getLastName() << ", " << getFirstName() << endl;
    cout << "-----" << endl;
    cout << (mHired ? "Current Employee" : "Former Employee") << endl;
    cout << "Employee Number: " << getEmployeeNumber() << endl;
    cout << "Salary: $" << getSalary() << endl;
    cout << endl;
}
```

A number of getters and setters perform the task of getting and setting values. Even though these methods seem trivial, it's better to have trivial getters and setters than to make your data members `public`. In the future, you may want to perform bounds checking in the `setSalary()` method, for example. Getters and setters also make debugging easier because you can put a breakpoint in them to inspect values when they are retrieved or set. Another reason is that when you decide to change how you are storing the data in your class, you would only need to modify these getters and setters.

```
// Getters and setters
void Employee::setFirstName(const string& firstName)
{
    mFirstName = firstName;
}
const string& Employee::getFirstName() const
{
    return mFirstName;
}
// ... other getters and setters omitted for brevity
}
```

EmployeeTest.cpp

As you write individual classes, it is often useful to test them in isolation. The following code includes a `main()` function that performs some simple operations using the `Employee` class. Once you are confident that the `Employee` class works, you should remove or comment-out this file so that you don't attempt to compile your code with multiple `main()` functions.

```
#include <iostream>
#include "Employee.h"
using namespace std;
using namespace Records;
int main()
{
    cout << "Testing the Employee class." << endl;
    Employee emp;
    emp.setFirstName("John");
    emp.setLastName("Doe");
    emp.setEmployeeNumber(71);
    emp.setSalary(50000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();
    return 0;
}
```

The Database Class

The Database class uses the `std::vector` class from the standard library to store Employee objects.

Database.h

Because the database will take care of automatically assigning an employee number to a new employee, a constant defines where the numbering begins.

```
#pragma once
#include <iostream>
#include <vector>
#include "Employee.h"
namespace Records {
    const int kFirstEmployeeNumber = 1000;
```

The database provides an easy way to add a new employee by providing a first and last name. For convenience, this method returns a reference to the new employee. External code can also get an employee reference by calling the `getEmployee()` method. Two versions of this method are declared. One allows retrieval by employee number. The other requires a first and last name.

```
Employee& getEmployee(int employeeNumber);
Employee& getEmployee(const std::string& firstName,
                      const std::string& lastName);
```

Because the database is the central repository for all employee records, it has methods that output all employees, the employees who are currently hired, and the employees who are no longer hired.

```
void displayAll() const;
void displayCurrent() const;
void displayFormer() const;
```

`mEmployees` contains the `Employee` objects. The `mNextEmployeeNumber` data member keeps track of what employee number is assigned to a new employee.

```
private:
    std::vector<Employee> mEmployees;
    int mNextEmployeeNumber;
};
```

Database.cpp

The `Database` constructor takes care of initializing the next employee number data member to its starting value.

```
#include <iostream>
#include <stdexcept>
#include "Database.h"
using namespace std;
namespace Records {
    Database::Database() : mNextEmployeeNumber(kFirstEmployeeNumber)
    {
    }
}
```

The `addEmployee()` method creates a new `Employee` object, fills in its information and adds it to the vector. Note that the `mNextEmployeeNumber` data member is incremented after its use so that the next employee will get a new number.

```
Employee& Database::addEmployee(const string& firstName,
                                 const string& lastName)
{
    Employee theEmployee;
    theEmployee.setFirstName(firstName);
    theEmployee.setLastName(lastName);
    theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
    theEmployee.hire();
    mEmployees.push_back(theEmployee);
    return mEmployees[mEmployees.size() - 1];
}
```

Only one version of `getEmployee()` is shown. Both versions work in similar ways. The methods loop over all employees in `mEmployees` using range-based `for` loops, and check to see if each `Employee` is a match for the information passed to the method. An exception is thrown if no match is found.

```

Employee& Database::getEmployee(int employeeNumber)
{
    for (auto& employee : mEmployees) {
        if (employee.getEmployeeNumber() == employeeNumber) {
            return employee;
        }
    }
    throw runtime_error("No employee found.");
}

```

The display methods all use a similar algorithm. They loop through all employees and tell each employee to display itself to the console if the criterion for display matches. `displayFormer()` is similar to `displayCurrent()`.

```

void Database::displayAll() const
{
    for (const auto& employee : mEmployees) {
        employee.display();
    }
}
void Database::displayCurrent() const
{
    for (const auto& employee : mEmployees) {
        if (employee.getIsHired())
            employee.display();
    }
}

```

DatabaseTest.cpp

A simple test for the basic functionality of the database follows:

```

#include <iostream>
#include "Database.h"
using namespace std;
using namespace Records;
int main()
{
    Database myDB;
    Employee& emp1 = myDB.addEmployee("Greg", "Wallis");
    emp1.fire();
    Employee& emp2 = myDB.addEmployee("Marc", "Gregoire");
    emp2.setSalary(100000);
    Employee& emp3 = myDB.addEmployee("John", "Doe");
    emp3.setSalary(10000);
    emp3.promote();
    cout << "all employees: " << endl << endl;
    myDB.displayAll();
    cout << endl << "current employees: " << endl << endl;
    myDB.displayCurrent();
    cout << endl << "former employees: " << endl << endl;
    myDB.displayFormer();
    return 0;
}

```

The User Interface

The final part of the program is a menu-based user interface that makes it easy for users to work with the employee database.

UserInterface.cpp

The `main()` function is a loop that displays the menu, performs the selected action, then does it all again. For most actions, separate functions are defined. For simpler actions, like displaying employees, the actual code is put in the appropriate case.

```
#include <iostream>
#include <stdexcept>
#include <exception>
#include "Database.h"
using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& db);
void doFire(Database& db);
void doPromote(Database& db);
void doDemote(Database& db);

int main()
{
    Database employeeDB;
    bool done = false;
    while (!done) {
        int selection = displayMenu();
        switch (selection) {
        case 1:
            doHire(employeeDB);
            break;
        case 2:
            doFire(employeeDB);
            break;
        case 3:
            doPromote(employeeDB);
            break;
        case 4:
            employeeDB.displayAll();
            break;
        case 5:
            employeeDB.displayCurrent();
            break;
        case 6:
            employeeDB.displayFormer();
            break;
        case 0:
            done = true;
            break;
        default:
            cerr << "Unknown command." << endl;
        }
    }
}
```

```

        break;
    }
}
return 0;
}

```

The `displayMenu()` function outputs the menu and gets input from the user. One important note is that this code assumes that the user will “play nice” and type a number when a number is requested. When you read about I/O in Chapter 12, you will learn how to protect against bad input.

```

int displayMenu()
{
    int selection;
    cout << endl;
    cout << "Employee Database" << endl;
    cout << "-----" << endl;
    cout << "1) Hire a new employee" << endl;
    cout << "2) Fire an employee" << endl;
    cout << "3) Promote an employee" << endl;
    cout << "4) List all employees" << endl;
    cout << "5) List all current employees" << endl;
    cout << "6) List all former employees" << endl;
    cout << "0) Quit" << endl;
    cout << endl;
    cout << "---> ";
    cin >> selection;
    return selection;
}

```

The `doHire()` function gets the new employee’s name from the user and tells the database to add the employee. It handles errors somewhat gracefully by outputting a message and continuing.

```

void doHire(Database& db)
{
    string firstName;
    string lastName;
    cout << "First name? ";
    cin >> firstName;
    cout << "Last name? ";
    cin >> lastName;
    try {
        db.addEmployee(firstName, lastName);
    } catch (const std::exception& exception) {
        cerr << "Unable to add new employee: " << exception.what() << endl;
    }
}

```

`doFire()` and `doPromote()` both ask the database for an employee by their employee number and then use the public methods of the `Employee` object to make changes.

```

void doFire(Database& db)
{
    int employeeNumber;
    cout << "Employee number? ";
    cin >> employeeNumber;
    try {

```

```
Employee& emp = db.getEmployee(employeeNumber);
emp.fire();
cout << "Employee " << employeeNumber << " terminated." << endl;
} catch (const std::exception& exception) {
    cerr << "Unable to terminate employee: " << exception.what() << endl;
}
}

void doPromote(Database& db)
{
    int employeeNumber;
    int raiseAmount;
    cout << "Employee number? ";
    cin >> employeeNumber;
    cout << "How much of a raise? ";
    cin >> raiseAmount;
    try {
        Employee& emp = db.getEmployee(employeeNumber);
        emp.promote(raiseAmount);
    } catch (const std::exception& exception) {
        cerr << "Unable to promote employee: " << exception.what() << endl;
    }
}
```

Evaluating the Program

The preceding program covers a number of topics from the very simple to the more complex. There are a number of ways that you could extend this program. For example, the user interface does not expose all of the functionality of the `Database` or `Employee` classes. You could modify the UI to include those features. You could also change the `Database` class to remove fired employees from `mEmployees`.

If there are parts of this program that don't make sense, consult the preceding sections to review those topics. If something is still unclear, the best way to learn is to play with the code and try things out. For example, if you're not sure how to use the conditional operator, write a short `main()` function that tries it out.

SUMMARY

Now that you know the fundamentals of C++, you are ready to become a professional C++ programmer. When you start getting deeper into the C++ language farther in the book, refer to this chapter to brush up on parts of the language you may need to review. Going back to some of the sample code in this chapter may be all you need to see to bring a forgotten concept back to the forefront of your mind.

2

Working with Strings

WHAT'S IN THIS CHAPTER?

- The differences between C-style strings and C++ strings
- Details of the C++ `std::string` class
- What raw string literals are

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Every program that you write will use strings of some kind. With the old C language there is not much choice but to use a dumb null-terminated character array to represent a string. Unfortunately, doing so can cause a lot of problems, such as buffer overflows, which can result in security vulnerabilities. The C++ STL includes a safe and easy-to-use `std::string` class that does not have these disadvantages.

This chapter discusses strings in more detail. It starts with a discussion of the old C-style strings, explains their disadvantages, and ends with the C++ `string` class and raw string literals.

DYNAMIC STRINGS

Strings in languages that have supported them as first-class objects tend to have a number of attractive features, such as being able to expand to any size, or have sub-strings extracted or replaced. In other languages, such as C, strings were almost an afterthought; there was no really good “string” data type, just fixed arrays of bytes. The “string library” was nothing

more than a collection of rather primitive functions without even bounds checking. C++ provides a string type as a first-class data type.

C-Style Strings

In the C language, strings are represented as an array of characters. The last character of a string is a null character ('\0') so that code operating on the string can determine where it ends. This null character is officially known as **NUL**, spelled with one L, not two. **NUL** is not the same as the **NULL** pointer. Even though C++ provides a better string abstraction, it is important to understand the C technique for strings because they still arise in C++ programming. One of the most common situations is where a C++ program has to call a C-based interface in some third-party library or as part of interfacing to the operating system.

By far, the most common mistake that programmers make with C strings is that they forget to allocate space for the '\0' character. For example, the string "hello" appears to be five characters long, but six characters worth of space are needed in memory to store the value, as shown in Figure 2-1.

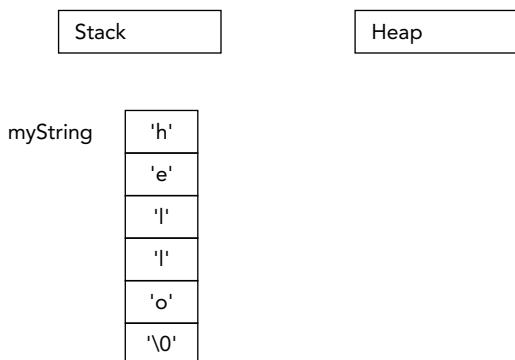


FIGURE 2-1

C++ contains several functions from the C language that operate on strings. These functions are defined in the `<cstring>` header. As a general rule of thumb, these functions do not handle memory allocation. For example, the `strcpy()` function takes two strings as parameters. It copies the second string onto the first, whether it fits or not. The following code attempts to build a wrapper around `strcpy()` that allocates the correct amount of memory and returns the result, instead of taking in an already allocated string. It uses the `strlen()` function to obtain the length of the string. The caller is responsible for freeing the memory allocated by `copyString()`.

```
char* copyString(const char* str)
{
    char* result = new char[strlen(str)]; // BUG! Off by one!
    strcpy(result, str);
    return result;
}
```

The `copyString()` function as written is incorrect. The `strlen()` function returns the length of the string, not the amount of memory needed to hold it. For the string "hello", `strlen()` will return 5, not 6. The proper way to allocate memory for a string is to add one to the amount of space needed for the actual characters. It seems a bit unnatural to have +1 all over the place. Unfortunately, that's how it works, so keep this in mind when you work with C-style strings. The correct implementation is as follows:

```
char* copyString(const char* str)
{
    char* result = new char[strlen(str) + 1];
    strcpy(result, str);
    return result;
}
```

One way to remember that `strlen()` returns only the number of actual characters in the string is to consider what would happen if you were allocating space for a string made up of several others. For example, if your function took in three strings and returned a string that was the concatenation of all three, how big would it be? To hold exactly enough space, it would be the length of all three strings, added together, plus one for the trailing '\0' character. If `strlen()` included the '\0' in the length of the string, the allocated memory would be too big. The following code uses the `strcpy()` and `strcat()` functions to perform this operation. The `cat` in `strcat()` stands for concatenate.

```
char* appendStrings(const char* str1, const char* str2, const char* str3)
{
    char* result = new char[strlen(str1) + strlen(str2) + strlen(str3) + 1];
    strcpy(result, str1);
    strcat(result, str2);
    strcat(result, str3);
    return result;
}
```

The `sizeof()` operator in C and C++ can be used to get the size of a certain data type or variable. For example, `sizeof(char)` returns 1 because a `char` has a size of 1 byte. However, in the context of C-style strings, `sizeof()` is not the same as `strlen()`. You should never use `sizeof()` to try to get the size of a string. If the C-style string is stored as a `char[]`, then `sizeof()` returns the actual memory used by the string, including the '\0' character. For example:

```
char text1[] = "abcdef";
size_t s1 = sizeof(text1); // is 7
size_t s2 = strlen(text1); // is 6
```

However, if the C-style string is stored as a `char*`, then `sizeof()` returns the size of a pointer! For example:

```
const char* text2 = "abcdef";
size_t s3 = sizeof(text2); // is platform-dependent
size_t s4 = strlen(text2); // is 6
```

`s3` will be 4 when compiled in 32-bit mode and will be 8 when compiled in 64-bit mode because it is returning the size of a `const char*`, which is a pointer.

A complete list of C functions to operate on strings can be found in the `<cstring>` header file.

WARNING When you use the C-style string functions with Microsoft Visual Studio, the compiler is likely to give you security-related warnings or even errors about these functions being deprecated. You can eliminate these warnings by using other C standard library functions, such as `strcpy_s()` or `strcat_s()`, which are part of the “secure C library” standard (ISO/IEC TR 24731). However, the best solution is to switch to the C++ string class, discussed later in this chapter.

String Literals

You’ve probably seen strings written in a C++ program with quotes around them. For example, the following code outputs the string `hello` by including the string itself, not a variable that contains it:

```
cout << "hello" << endl;
```

In the preceding line, “`hello`” is a *string literal* because it is written as a value, not a variable. The actual memory associated with a string literal is in a read-only part of memory. This allows the compiler to optimize memory usage by reusing references to equivalent string literals. That is, even if your program uses the string literal “`hello`” 500 times, the compiler is allowed to create just one instance of `hello` in memory. This is called *literal pooling*.

String literals can be *assigned* to variables, but because string literals are in a read-only part of memory and because of the possibility of literal pooling, assigning them to variables can be risky. The C++ standard officially says that string literals are of type “array of n `const char`”; however, for backward compatibility with older non-`const` aware code, most compilers do not enforce your program to assign a string literal to a variable of type `const char*`. They let you assign a string literal to a `char*` without `const`, and the program will work fine unless you attempt to change the string. Generally, the behavior of modifying string literals is undefined. It could, for example, cause a crash, or it could keep working with seemingly inexplicable side effects, or the modification could silently be ignored, or it could just work; it all depends on your compiler. For example, the following code exhibits undefined behavior:

```
char* ptr = "hello";           // Assign the string literal to a variable.
ptr[1] = 'a';                 // Undefined behavior!
```

A much safer way to code is to use a pointer to `const` characters when referring to string literals. The following code contains the same bug, but because it assigned the literal to a `const char*`, the compiler will catch the attempt to write to read-only memory.

```
const char* ptr = "hello"; // Assign the string literal to a variable.
ptr[1] = 'a';             // Error! Attempts to write to read-only memory
```

You can also use a string literal as an initial value for a character array (`char[]`). In this case, the compiler creates an array that is big enough to hold the string and copies the string to this array. So, the compiler will not put the literal in read-only memory and will not do any literal pooling.

```
char arr[] = "hello"; // Compiler takes care of creating appropriate sized
                      // character array arr.
arr[1] = 'a';         // The contents can be modified.
```

The C++ string Class

C++ provides a much-improved implementation of the concept of a string as part of the Standard Library. In C++, `std::string` is a class (actually an instantiation of the `basic_string` class template) that supports many of the same functionalities as the `<cstring>` functions, but takes care of memory allocation for you. The `string` class is defined in the `<string>` header in the `std` namespace, and has already been introduced in the previous chapter. Now it's time to take a deeper look at it.

What Is Wrong with C-Style Strings?

To understand the necessity of the C++ `string` class, consider the advantages and disadvantages of C-style strings.

Advantages:

- They are simple, making use of the underlying basic character type and array structure.
- They are lightweight, taking up only the memory that they need if used properly.
- They are low level, so you can easily manipulate and copy them as raw memory.
- They are well understood by C programmers — why learn something new?

Disadvantages:

- They require incredible efforts to simulate a first-class string data type.
- They are unforgiving and susceptible to difficult-to-find memory bugs.
- They don't leverage the object-oriented nature of C++.
- They require knowledge of their underlying representation on the part of the programmer.

The preceding lists were carefully constructed to make you think that perhaps there is a better way. As you'll learn, C++ strings solve all the problems of C strings and render most of the arguments about the advantages of C strings over a first-class data type irrelevant.

Using the `string` Class

Even though `string` is a class, you can almost always treat it as if it were a built-in type. In fact, the more you think of it as a simple type, the better off you are. Through the magic of operator overloading, C++ strings are much easier to use than C-style strings. For example, the `+` operator is redefined for strings to mean “string concatenation.” The following produces 1234:

```
string A("12");
string B("34");
string C;
C = A + B;      // C will become "1234"
```

The `+=` operator is also overloaded to allow you to easily append a string:

```
string A("12");
string B("34");
A += B;      // A will become "1234"
```

Another problem with C strings is that you cannot use `==` to compare them. Suppose you have the following two strings:

```
char* a = "12";
char b[] = "12";
```

Writing a comparison as follows always returns `false`, because it compares the pointer values, not the contents of the strings:

```
if (a == b)
```

Note that C arrays and pointers are related. You can think of C arrays, like the `b` array in the example, as pointers to the first element in the array. Chapter 22 goes deeper in on the array-pointer duality.

To compare C strings, you have to write something as follows:

```
if (strcmp(a, b) == 0)
```

Furthermore, there is no way to use `<`, `<=`, `>=`, or `>` to compare C strings, so `strcmp()` returns `-1`, `0`, or `1` depending on the lexicographic relationship of the strings. This results in very clumsy code, which is also error-prone.

With C++ strings, `operator==`, `operator!=`, `operator<`, and so on are all overloaded to work on the actual string characters. Individual characters can still be accessed with `operator[]`.

As the following code shows, when string operations require extending the `string`, the memory requirements are automatically handled by the `string` class, so memory overruns are a thing of the past.

```
string myString = "hello";
myString += ", there";
string myOtherString = myString;
if (myString == myOtherString) {
    myOtherString[0] = 'H';
}
cout << myString << endl;
cout << myOtherString << endl;
```

The output of this code is:

```
hello, there
Hello, there
```

There are several things to note in this example. One point to note is that there are no memory leaks even though strings are allocated and resized left and right. All of these `string` objects are created as stack variables. While the `string` class certainly has a bunch of allocating and resizing to do, the `string` destructors clean up this memory when `string` objects go out of scope.

Another point to note is that the operators work the way you want them to. For example, the `=` operator copies the strings, which is most likely what you want. If you are used to working with array-based strings, this will either be refreshingly liberating for you or somewhat confusing. Don't worry — once you learn to trust the `string` class to do the right thing, life gets so much easier.

For compatibility, you can use the `c_str()` method on a `string` to get a `const` character pointer, representing a C-style string. However, the returned `const` pointer becomes invalid whenever the

`string` has to perform any memory reallocation, or when the `string` object is destroyed. You should call the method just before using the result so that it accurately reflects the current contents of the `string`, and you must never return the result of `c_str()` called on a stack-based `string` from your function.

Consult a Standard Library Reference, for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>, for a complete list of all supported operations that you can perform on `string` objects.

std::string Literals



A string literal in source code is usually interpreted as a `const char*`. You can use the standard user-defined literal "`s`" to interpret a string literal as an `std::string` instead. For example:

```
auto string1 = "Hello World";      // string1 will be a const char*
auto string2 = "Hello World"s;     // string2 will be an std::string
```

Numeric Conversions

The `std` namespace includes a number of helper functions making it easy to convert numerical values into strings or strings into numerical values. The following functions are available to convert numerical values into strings:

- `string to_string(int val);`
- `string to_string(unsigned val);`
- `string to_string(long val);`
- `string to_string(unsigned long val);`
- `string to_string(long long val);`
- `string to_string(unsigned long long val);`
- `string to_string(float val);`
- `string to_string(double val);`
- `string to_string(long double val);`

They are pretty straightforward to use. For example, the following code converts a `long double` value into a `string`:

```
long double d = 3.14L;
string s = to_string(d);
```

Converting in the other direction is done by the following set of functions, also defined in the `std` namespace. In these prototypes, `str` is the `string` that you want to convert, `idx` is a pointer that will receive the index of the first non-converted character, and `base` is the mathematical base that should be used during conversion. The `idx` pointer can be a null pointer in which case it will be ignored. These functions throw `invalid_argument` if no conversion could be performed and throw `out_of_range` if the converted value is outside the range of the return type.

```
➤  int stoi(const string& str, size_t *idx=0, int base=10);
➤  long stol(const string& str, size_t *idx=0, int base=10);
➤  unsigned long stoul(const string& str, size_t *idx=0, int base=10);
➤  long long stoll(const string& str, size_t *idx=0, int base=10);
➤  unsigned long long stoull(const string& str, size_t *idx=0, int base=10);
➤  float stof(const string& str, size_t *idx=0);
➤  double stod(const string& str, size_t *idx=0);
➤  long double stold(const string& str, size_t *idx=0);
```

For example:

```
const string s = "1234";
int i = stoi(s);      // i will be 1234
```

Raw String Literals

Raw string literals are string literals that can span across multiple lines of code, that don't require escaping of embedded double quotes, and where escape sequences like \t and \n are not processed as escape sequences, but as normal text. Escape sequences are discussed in Chapter 1. For example, if you write the following with a normal string literal, you will get a compiler error because the string contains non-escaped double quotes:

```
string str = "Hello "World"!";      // Error!
```

With a normal string you have to escape the double quotes as follows:

```
string str = "Hello \"World\"!";
```

With a raw string literal you can avoid the need to escape the quotes. The raw string literal starts with R"(and ends with)".

```
string str = R"(Hello "World"!);
```

Raw string literals can span across multiple lines. For example, if you write the following with a normal string literal, you will get a compiler error, because a normal string literal cannot span multiple lines:

```
string str = "Line 1
Line 2 with \t";      // Error!
```

Instead, you can use a raw string literal as follows:

```
string str = R"(Line 1
Line 2 with \t);
```

This also demonstrates that with the raw string literal the \t escape character is not replaced with an actual tab character but is taken literally. If you write str to the console the output will be:

```
Line 1
Line 2 with \t
```

Since the raw string literal ends with `)` " you cannot embed a `)` " in your string using this syntax. For example, the following string is not valid because it contains the `)` " in the middle of the string:

```
string str = R"(The characters )" are embedded in this string); // Error!
```

If you need embedded `)` " characters, you need to use the extended raw string literal syntax, which is as follows:

```
R"d-char-sequence(r-char-sequence)d-char-sequence"
```

The `r-char-sequence` is the actual raw string. The `d-char-sequence` is an optional delimiter sequence, which should be the same at the beginning and at the end of the raw string literal. This delimiter sequence can have at most 16 characters. You should choose this delimiter sequence as a sequence that will not appear in the middle of your raw string literal.

The previous example can be rewritten using a unique delimiter sequence as follows:

```
string str = R"- (The characters )" are embedded in this string) -";
```

Raw string literals make it easier to work with database querying strings, regular expressions, and so on. Regular expressions are discussed in Chapter 18.

Nonstandard Strings

There are several reasons why many C++ programmers don't use C++-style strings. Some programmers simply aren't aware of the `string` type because it was not always part of the C++ specification. Others have discovered over the years that the C++ `string` doesn't provide the behavior they need and have developed their own string type. Perhaps the most common reason is that development frameworks and operating systems tend to have their own way of representing strings, such as the `CString` class in Microsoft's MFC. Often, this is for backward compatibility or legacy issues. When starting a project in C++, it is very important to decide ahead of time how your group will represent strings.

SUMMARY

This chapter discussed the C++ `string` class and why you should use it instead of the old plain C-style character arrays. It also explained a number of helper functions to make it easier to convert numerical values into `strings` and vice versa, and introduced the concept of raw string literals.

3

Coding with Style

WHAT'S IN THIS CHAPTER?

- The importance of documenting your code, and what kind of commenting styles you can use
- What decomposition means and how to use it
- What naming conventions are
- What formatting rules are

If you're going to spend several hours each day in front of a keyboard writing code, you should take some pride in all that work. Writing code that gets the job done is only part of a programmer's work. After all, anybody can learn the fundamentals of coding. It takes a true master to code with style.

This chapter explores the question of what makes good code. Along the way, you'll see several approaches to C++ style. As you will discover, simply changing the style of code can make it appear very different. For example, C++ code written by Windows programmers often has its own style, using Windows conventions. It almost looks like a completely different language than C++ code written by Mac OS programmers. Exposure to several different styles will help you avoid that sinking feeling you get when opening up a C++ source file that barely resembles the C++ you thought you knew.

THE IMPORTANCE OF LOOKING GOOD

Writing code that is stylistically “good” takes time. You probably don't need much time to whip together a quick-and-dirty program to parse an XML file. Writing the same program with functional decomposition, adequate comments, and a clean structure would take you more time. Is it really worth it?

Thinking Ahead

How confident would you be in your code if a new programmer had to work with it a year from now? A friend of mine, faced with a growing mess of web application code, encouraged his team to think about a hypothetical intern who would be starting in a year. How would this poor intern ever get up to speed on the code base when there was no documentation and scary multiple-page functions? When you're writing code, imagine that somebody new will have to maintain it in the future. Will you even remember how it works? What if you're not available to help? Well-written code avoids these problems because it is easy to read and understand.

Elements of Good Style

It is difficult to enumerate the characteristics of code that make it “stylistically good.” Over time, you’ll find styles that you like and notice useful techniques in code that others wrote. Perhaps more important, you’ll encounter horrible code that teaches you what to avoid. However, good code shares several universal tenets that are explored in this chapter.

- Documentation
- Decomposition
- Naming
- Use of the Language
- Formatting

DOCUMENTING YOUR CODE

In the programming context, documentation usually refers to comments contained in the source files. Comments are your opportunity to tell the world what was going through your head when you wrote the accompanying code. They are a place to say anything that isn’t obvious from looking at the code itself.

Reasons to Write Comments

It may seem obvious that writing comments is a good idea, but have you ever stopped to think about why you need to comment your code? Sometimes programmers recognize the importance of commenting without fully understanding why comments are important. There are several reasons, all of which are explored in this chapter.

Commenting to Explain Usage

One reason to use comments is to explain how clients should interact with the code. Each publicly accessible function or method in a header file should have a comment explaining what it does. Some organizations prefer to formalize these comments by explicitly listing the purpose of each method, what its arguments are, what values it returns, and possible exceptions it can throw.

Providing a comment with public methods accomplishes two things. First, you are given the opportunity to state, in English, anything that you can't state in code. For example, there's really no way in C++ code to indicate that the `saveRecord()` method of a database object can only be called after the `openDatabase()` method is called. A comment, however, can be the perfect place to note this restriction, as follows.

```
/*
 * saveRecord()
 *
 * Saves the given record to the database.
 *
 * This method will throw a "DatabaseNotOpenedException"
 * if the openDatabase() method was not called first.
 */
```

The second effect of a comment on a public method can be to state usage information. The C++ language forces you to specify the return type of a method, but it does not provide a way for you to say what the returned value actually represents. For example, the declaration of the `saveRecord()` method may indicate that it returns an `int`, but the client reading that declaration wouldn't know what the `int` means. Other ancillary data can be included in a comment as well, as shown in the following example.

```
/*
 * saveRecord()
 *
 * Saves the given record to the database.
 *
 * Parameters:
 *     Record& rec: the record to save to the database.
 * Returns:
 *     An integer representing the ID of the saved record.
 * Throws:
 *     DatabaseNotOpenedException if the openDatabase() method was not
 *     called first.
 */
```

Sometimes, the parameters to, and the return type from a function are generic and can be used to pass all kinds of information. In that case you need to clearly document what exact type is being passed. For example, message handlers in Windows accept two parameters, `LPARAM` and `WPARAM`, and can return an `LRESULT`. All three can be used to pass anything you like, but you cannot change the type of them. By using type casting, they can for example be used to pass a simple integer or to pass a pointer to some object. Your documentation could look as follows.

```
* Parameters:
*     WPARAM wParam: (WPARAM) (int): An integer representing...
*     LPARAM lParam: (LPARAM) (string*): A string representing...
* Returns:
*     nullptr in case of an error, otherwise a pointer to a Record object
*     representing...
```

Most editors allow you to bind keystrokes to perform certain actions. You could bind a keystroke so that the editor automatically inserts a standard commenting block which you subsequently fill in with the right information. For example, the keystroke could automatically insert the following comment template.

```
/*
 * func()
 *
 * Description of the function.
 *
 * Parameters:
 *   int param1: parameter 1.
 * Returns: int
 *   An integer representing...
 * Throws:
 *   Exception1 if...
 * Notes:
 *   Additional notes...
 */
```

Commenting to Explain Complicated Code

Good comments are also important inside the actual source code. In a simple program that processes input from the user and writes a result to the console, it is probably easy to read through and understand all of the code. In the professional world, however, you will often need to write code that is algorithmically complex or too esoteric to understand simply by inspection.

Consider the code that follows. It is well written, but it may not be immediately apparent what it is doing. You might recognize the algorithm if you have seen it before, but a newcomer probably wouldn't understand the way the code works.

```
void sort(int inArray[], int inSize)
{
    for (int i = 1; i < inSize; i++) {
        int element = inArray[i];
        int j = i - 1;
        while (j >= 0 && inArray[j] > element) {
            inArray[j+1] = inArray[j];
            j--;
        }
        inArray[j+1] = element;
    }
}
```

A better approach would be to include comments that describe the algorithm that is being used, and to document (loop) invariants. Invariants are conditions that have to be true during the execution of a piece of code, for example a loop iteration. In the modified function that follows, a thorough comment at the top explains the algorithm at a high level, and inline comments explain specific lines that may be confusing.

```
/*
 * Implements the "insertion sort" algorithm. The algorithm separates the
 * array into two parts--the sorted part and the unsorted part. Each
```

```

* element, starting at position 1, is examined. Everything earlier in the
* array is in the sorted part, so the algorithm shifts each element over
* until the correct position is found for the current element. When the
* algorithm finishes with the last element, the entire array is sorted.
*/
void sort(int inArray[], int inSize)
{
    // Start at position 1 and examine each element.
    for (int i = 1; i < inSize; i++) {
        // Invariant: All elements in the range 0 to i-1 (inclusive) are sorted.
        int element = inArray[i];
        // j marks the position in the sorted part of the array.
        int j = i - 1;
        // As long as the current slot in the sorted array is higher than
        // the element, shift the slot over and move backwards.
        while (j >= 0 && inArray[j] > element) {
            inArray[j+1] = inArray[j];
            j--;
        }
        // At this point the current position in the sorted array
        // is *not* greater than the element, so this is its new position.
        inArray[j+1] = element;
    }
}

```

The new code is certainly more verbose, but a reader unfamiliar with sorting algorithms would be much more likely to understand it with the comments included. In some organizations, inline comments are frowned upon. In such cases, writing clean code and having good comments at the top of the function becomes vital.

Commenting to Convey Metainformation

Another reason to use comments is to provide information at a higher level than the code itself. This *metainformation* provides details about the creation of the code without addressing the specifics of its behavior. For example, your organization may want to keep track of the original author of each method. You can also use metainformation to cite external documents or refer to other code.

The following example shows several instances of metainformation, including the author of the file, the date it was created, and the specific feature it addresses. It also includes inline comments expressing metadata, such as the bug number that corresponds to a line of code and a reminder to revisit a possible problem in the code later.

```

/*
 * Author:  marcg
 * Date:    110412
 * Feature: PRD version 3, Feature 5.10
 */
int saveRecord(Record& rec)
{
    if (!mDatabaseOpen) {
        throw DatabaseNotOpenedException();
    }

```

```
int id = getDB() -> saveRecord(rec);
if (id == -1) return -1; // Added to address bug #142 - jsmith 110428
rec.setId(id);
// TODO: What if setId() throws an exception? - akshayr 110501
return id;
}
```

A change-log could also be included at the beginning of each file. The following shows a possible example of such a change-log.

```
/*
 * Date      | Change
 *-----+-----+
 * 110413  | REQ #005: <marcg> Do not normalize values.
 * 110417  | REQ #006: <marcg> use nullptr instead of NULL.
 */
```

However, this might not be necessary when you use a source code control solution, discussed in Chapter 24. They offer an annotated change history with revision dates, authors, and, if properly used, comments accompanying each modification, including references to change requests. You should check-in each change request separately. For example, if you need to implement two change requests in one file, you should check-out the file, implement the first change request and check-in the file with the appropriate change-log comment. Only then you can check-out the file again and work on the second change request. If you want to work on both change requests at the same time, you can branch the source file and start working on the first change request in one branch and work on the second change request in the second branch. When implementation is finished, you can merge both branches back together with appropriate change-log comments for each branch. With this method you don't need to manually keep a change-log in the beginning of each file.

It's easy to go overboard with comments. A good approach is to discuss which types of comments are most useful with your group and form a policy. For example, if one member of the group uses a "TODO" comment to indicate code that still needs work, but nobody else knows about this convention, the code in need could be overlooked.

NOTE *If your group decides to use metainformation comments, make sure that you all include the same information or your files will be inconsistent.*

Commenting Styles

Every organization has a different approach to commenting code. In some environments, a particular style is mandated to give the code a common standard for documentation. Other times, the quantity and style of commenting is left up to the programmer. The following examples depict several approaches to commenting code.

Commenting Every Line

One way to avoid lack of documentation is to force yourself to overdocument by including a comment for every line. Commenting every line of code should ensure that there's a specific reason

for everything you write. In reality, such heavy commenting on a large-scale basis is unscalable, messy, and tedious. For example, consider the following useless comments.

```
int result;           // Declare an integer to hold the result.
result = doodad.getResult(); // Get the doodad's result.
if (result % 2 == 0) { // If the result mod 2 is 0 ...
    logError();       // then log an error,
} else {             // otherwise ...
    logSuccess();    // log success.
}
return result;       // End if/else
// Return the result
```

The comments in this code express each line as part of an easily readable English story. This is entirely useless if you assume that the reader has at least basic C++ skills. These comments don't add any additional information to code. Specifically, look at this line:

```
if (result % 2 == 0) { // If the result mod 2 is 0 ...
```

The comment is just an English translation of the code. It doesn't say *why* the programmer has used the mod operator on the result with the value 2. A better comment would be:

```
if (result % 2 == 0) { // If the result is even ...
```

The modified comment, while still fairly obvious to most programmers, gives additional information about the code. The result is "modded" by 2 because the code needs to check if the result is even.

Despite its tendency to be verbose and superfluous, heavy commenting can be useful in cases where the code would otherwise be difficult to comprehend. The following code also comments every line, but these comments are actually helpful.

```
// Calculate the doodad. The start, end, and offset values come from the
// table on page 96 of the "Doodad API v1.6".
result = doodad.calculate(kStart, kEnd, kOffset);
// To determine success or failure, we need to bitwise AND the result with
// the processor-specific mask (see "Doodad API v1.6", page 201).
result &= getProcessorMask();
// Set the user field value based on the "Marigold Formula."
// (see "Doodad API v1.6", page 136)
setUserField((result + kMarigoldOffset) / MarigoldConstant + MarigoldConstant);
```

This code is taken out of context, but the comments give you a good idea of what each line does. Without them, the calculations involving & and the mysterious "Marigold Formula" would be difficult to decipher.

NOTE *Commenting every line of code is usually untenable, but if the code is complicated enough to require it, don't just translate the code to English: explain what's really going on.*

Prefix Comments

Your group may decide to begin all source files with a standard comment. This is an excellent opportunity to document important information about the program and specific file. Examples of information that you might want to document at the top of every file include the following.

- The last-modified date
- The original author
- A change-log as described earlier
- The feature ID addressed by the file
- Copyright information
- A brief description of the file/class
- Incomplete features
- Known bugs

Your development environment may allow you to create a template that automatically starts new files with your prefix comment. Some source control systems such as Subversion (SVN) can even assist by filling in metadata. For example, if your comment contains the string `Id`, SVN can automatically expand the comment to include the author, filename, revision, and date.

An example of a prefix comment is shown here:

```
/*
 * $Id: Watermelon.cpp,123 2004/03/10 12:52:33 marcg $
 *
 * Implements the basic functionality of a watermelon. All units are expressed
 * in terms of seeds per cubic centimeter. Watermelon theory is based on the
 * white paper "Algorithms for Watermelon Processing."
 *
 * The following code is (c) copyright 2011, FruitSoft, Inc. ALL RIGHTS RESERVED
 */
```

Fixed-Format Comments

Writing comments in a standard format that can be parsed by external document builders is an increasingly popular programming practice. In the Java language, programmers can write comments in a standard format that allows a tool called JavaDoc to create hyperlinked documentation for the project automatically. For C++, a free tool called Doxygen (available at www.doxygen.org) parses comments to automatically build HTML documentation, class diagrams, UNIX man pages, and other useful documents. Doxygen even recognizes and parses JavaDoc-style comments in C++ programs. The code that follows shows JavaDoc-style comments that are recognized by Doxygen.

```
/**
 * Implements the basic functionality of a watermelon
 * TODO: Implement updated algorithms!
 */
class Watermelon
{
```

```

public:
    /**
     * @param initialSeeds The starting number of seeds
     */
    Watermelon(int initialSeeds);
    /**
     * Computes the seed ratio, using the Marigold algorithm.
     * @param slowCalc Whether or not to use long (slow) calculations
     * @return The Marigold ratio
     */
    double calcSeedRatio(bool slowCalc);
};

```

Doxygen recognizes the C++ syntax and special comment directives such as `@param` and `@return` to generate customizable output. An example of a Doxygen-generated HTML class reference is shown in Figure 3-1.

Automatically generated documentation like the file shown in Figure 3-1 can be helpful during development because it allows developers to browse through a high-level description of classes and their relationships. Your group can easily customize a tool like Doxygen to work with the style of comments that you have adopted. Ideally, your group would set up a machine that builds documentation on a daily basis.

Ad Hoc Comments

Most of the time, you use comments on an as-needed basis. Here are some guidelines for comments that appear within the body of your code.

- Do your best to avoid offensive or derogatory language. You never know who might look at your code some day.
- Liberal use of inside jokes is generally considered okay. Check with your manager.
- Don't put your initials in the code. Source code control solutions will track that kind of information automatically for you.
- If you are doing something with an API that isn't immediately obvious, include a reference to the documentation of that API where it is explained.
- Remember to update your comments when you update the code. Nothing is more confusing than code that is fully documented with incorrect information.
- If you use comments to separate a function into sections, consider whether the function might be broken into multiple, smaller functions.

Main Page | Class List | File List | Class Members

Watermelon Class Reference

Implements the basic functionality of a watermelon. [More...](#)

```
#include <Watermelon.h>
```

[List of all members.](#)

Public Member Functions

```
Watermelon (int initialSeeds)
double calcSeedRatio (bool slowCalc)
    Computes the seed ratio, using the Marigold algorithm.
```

Detailed Description

TODO: Implement updated algorithms!

Definition at line 6 of file [Watermelon.h](#).

Constructor & Destructor Documentation

```
Watermelon::Watermelon (int initialSeeds)
```

Parameters:
`initialSeeds` The starting number of seeds

Member Function Documentation

```
double Watermelon::calcSeedRatio (bool slowCalc)
```

Parameters:
`slowCalc` Whether or not to use long (slow) calculations

FIGURE 3-1

Self-Documenting Code

Well-written code doesn't always need abundant commenting. The best code is written to be readable. If you find yourself adding a comment for every line, consider whether the code could be rewritten to better match what you are saying in the comments. For example, use descriptive names for your functions, parameters, variables, and so on. Remember that C++ is a language. Its main purpose is to tell the computer what to do, but the semantics of the language can also be used to explain its meaning to a reader.

Another way of writing self-documenting code is to break up, or *decompose*, your code into smaller pieces. Decomposition is covered in detail in the material that follows.

NOTE *Good code is naturally readable and only requires comments to provide useful additional information.*

Comments in This Book

The code examples you will see in this book often use comments to explain complicated code or to point things out to you that may not be evident.

DECOMPOSITION

Decomposition is the practice of breaking up code into smaller pieces. There is nothing more daunting in the world of coding than opening up a file of source code to find 300-line functions and massive nested blocks of code. Ideally, each function or method should accomplish a single task. Any subtasks of significant complexity should be decomposed into separate functions or methods. For example, if somebody asks you what a method does and you answer “First it does A, then it does B; then, if C, it does D; otherwise, it does E,” you should probably have separate helper methods for A, B, C, D, and E.

Decomposition is not an exact science. Some programmers will say that no function should be longer than a page of printed code. That may be a good rule of thumb, but you could certainly find a quarter-page of code that is desperately in need of decomposition. Another rule of thumb is if you squint your eyes and look at the format of the code without reading the actual content, it shouldn't appear too dense in any one area. For example, Figures 3-2 and 3-3 show code that has been purposely blurred so that you don't focus on the content. It should be obvious that the code in Figure 3-3 has better decomposition than the code in Figure 3-2.

```
void someFunction(int argc, char argv, Structure *arg3)
{
    arg3->sfef(argv, argv);
    dfef dfef;
    int sfefef;

    if (sfef && argv || argv->sfef) {
        dfef df = dfefdf;
        argv->sfefef(df);
        cout << argv;
        cerr << "sfef dfef dfef & dfef" << endl;
    } else {
        dfefef ad fadefdf;
        cout << argv->sfefef;
        cerr << "sfef" << endl;
    }

    // now do something else
    cout << "things" << argv->sfefef << endl;
    cout << "things" << argv->sfefef << endl;
    cout << "things" << argv->sfefef << endl;
}
```

FIGURE 3-2

```
void someFunction(int argc, char argv, Structure *arg3)
{
    arg3->sfef(argv, argv);

    if (sfefdf() {
        thing1();
    } else {
        thing2();
    }

    thing3();
}

bool sfefdf()
{
    return sfef & dfef & argv || argv->sfef;
}

void thing1()
{
    dfefdf df = dfefdf;
    argv->sfefef(df);
    cout << argv;
    cerr << "sfef dfef dfef & dfef" << endl;
}

void thing2()
{
    dfefdf ad fadefdf;
    cout << argv->sfefef;
    cerr << "sfef" << endl;
}

void thing3()
{
    cout << "things" << argv->sfefef << endl;
    cout << "things" << argv->sfefef << endl;
    cout << "things" << argv->sfefef << endl;
}
```

FIGURE 3-3

Decomposition through Refactoring

Sometimes when you've had a few coffees and you're really in the programming zone, you start coding so fast that you end up with code that does exactly what it's supposed to do, but is far from pretty. All programmers do this from time to time. Short periods of vigorous coding are sometimes the most productive times in the course of a project. Dense code also arises over the course of time as code is modified. As new requirements and bug fixes emerge, existing code is amended with small modifications. The computing term *cruft* refers to the gradual accumulation of small amounts of code that eventually turns a once-elegant piece of code into a mess of patches and special cases.

Refactoring is the act of restructuring your code. The following are a couple of example techniques that you can use to refactor your code. Consult a refactoring book listed in Appendix B to get a more comprehensive list.

- Techniques that allow for more abstraction:
 - **Encapsulate Field:** Make a field private and give access to it with getter and setter methods.
 - **Generalize Type:** Create more general types to allow for more code sharing.
- Techniques for breaking code apart into more logical pieces:
 - **Extract Method:** Turn part of a larger method into a new method to make it easier to understand.
 - **Extract Class:** Move part of the code from an existing class into a new class.
- Techniques for improving names and the location of code:
 - **Move Method or Move Field:** Move to a more appropriate class or source file.
 - **Rename Method or Rename Field:** Change the name to better reveal its purpose.
 - **Pull Up:** In object-oriented programming, move to a base class.
 - **Push Down:** In object-oriented programming, move to a derived class.

Whether your code starts its life as a dense block of unreadable cruft or it just evolves that way, refactoring is necessary to periodically purge the code of accumulated hacks. Through refactoring, you revisit existing code and rewrite it to make it more readable and maintainable. Refactoring is an opportunity to revisit the decomposition of code. If the purpose of the code has changed or if it was never decomposed in the first place, when you refactor the code, squint at it and determine if it needs to be broken down into smaller parts.

Decomposition by Design

If you use modular decomposition and approach every module, method, or function by considering what pieces of it you can put off until later, your programs will generally be less dense and more organized than if you implemented every feature in its entirety as you coded.

Of course, you should still do some design of your program *before* jumping into the code.

Decomposition in This Book

You will see decomposition in many of the examples in this book. In many cases, methods are referred to for which no implementation is shown because they are not relevant to the example and would take up too much space.

NAMING

The compiler has a couple of naming rules. Names cannot start with a number. You must also not use names that contain a double underscore (for example `my__name`) or names that begin with an underscore (for example `_Name`) because these are reserved for use by the compiler and the standard library implementation. Other than that, names exist only to help you and your fellow programmers work with the individual elements of your program. Given this purpose, it is surprising how often programmers use unspecific or inappropriate names in their programs.

Choosing a Good Name

The best name for a variable, method, function, or class accurately describes the purpose of the item. Names can also imply additional information, such as the type or specific usage. Of course, the real test is whether other programmers understand what you are trying to convey with a particular name.

There are no set-in-stone rules for naming other than the rules that work for your organization. However, there are some names that are rarely appropriate. The following table shows some names at the two extreme ends of the naming continuum.

GOOD NAMES	BAD NAMES
<code>sourceName, destinationName</code> Distinguishes two objects	<code>thing1, thing2</code> Too general
<code>gSettings</code> Conveys global status	<code>globalUserSpecificSettingsAndPreferences</code> Too long
<code>mNameCounter</code> Conveys data member status	<code>mNC</code> Too obscure, concise
<code>calculateMarigoldOffset()</code> Simple, accurate	<code>doAction()</code> Too general, imprecise
<code>mTypeString</code> Easy on the eyes	<code>typeSTR256</code> A name only a computer could love
<code>mWelshRarebit</code> Acceptable inside joke	<code>mIHateLarry</code> Unacceptable inside joke
<code>errorMessage</code> Descriptive name	<code>string</code> Non-descriptive name
<code>sourceFile, destinationFile</code> No abbreviations	<code>srcFile, dstFile</code> Abbreviations

Naming Conventions

Selecting a name doesn't always require a lot of thought and creativity. In many cases, you'll want to use standard techniques for naming. Following are some of the types of data for which you can make use of standard names.

Counters

Early in your programming career, you probably saw code that used the variable “i” as a counter. It is customary to use `i` and `j` as counters and inner-loop counters, respectively. Be careful with nested loops, however. It's a common mistake to refer to the “ith” element when you really mean the “jth” element. Some programmers prefer using counters like `outerLoopIndex` and `innerLoopIndex` instead, and some even frown upon using `i` and `j` as loop counters.

Prefixes

Many programmers begin their variable names with a letter that provides some information about the variable's type or usage. On the other hand, there are as many programmers who frown upon using any kind of prefix because they could make evolving code less maintainable in the future. For example, if a member variable is changed from `static` to non-`static`, this would mean that you have to rename all the uses of that name. This is often time consuming and so most programmers don't bother to rename the variables. As the code evolves, the declarations of the variables change but the names do not. This results in names giving the illusion of conveying semantics but in fact they convey the wrong semantics.

However, often you don't have a choice and you need to follow the guidelines for your company. The following table shows some potential prefixes.

PREFIX	EXAMPLE NAME	LITERAL PREFIX MEANING	USAGE
<code>m</code> <code>m_</code>	<code>mData</code> <code>m_data</code>	“member”	Data member within a class.
<code>s</code> <code>ms</code> <code>ms_</code>	<code>sLookupTable</code> <code>msLookupTable</code> <code>ms_lookupTable</code>	“static”	Static variable or data member.
<code>k</code>	<code>kMaximumLength</code>	“konstant” (German for “constant”)	A constant value. Some programmers use all uppercase names to indicate constants.
<code>b</code> <code>is</code>	<code>bCompleted</code> <code>isCompleted</code>	“Boolean”	Designates a Boolean value.
<code>n</code> <code>mNum</code>	<code>nLines</code> <code>mNumLines</code>	“number”	A data member that is also a counter. Since an “n” looks similar to an “m,” some programmers instead use <code>mNum</code> as a prefix, as in <code>mNumLines</code> .

Getters and Setters

If your class contains a data member, such as `mStatus`, it is customary to provide access to the member via a getter called `getStatus()` and a setter called `setStatus()`. The C++ language has no prescribed naming for these methods, but your organization will probably want to adopt this or a similar naming scheme.

Capitalization

There are many different ways of capitalizing names in your code. As with most elements of coding style, the most important thing is that your group standardizes on an approach and that all members adopt that approach. One way to get messy code is to have some programmers naming classes in all lowercase with underscores representing spaces (`priority_queue`) and others using capitals with each subsequent word capitalized (`PriorityQueue`). Variables and data members almost always start with a lowercase letter and either use underscores (`my_queue`) or capitals (`myQueue`) to indicate word breaks. Functions and methods are traditionally capitalized in C++, but, as you've seen, in this book I have adopted the style of lowercase functions and methods to distinguish them from class names. A similar style of capitalizing letters is used to indicate word boundaries for class and data member names.

Namespaced Constants

Imagine that you are writing a program with a graphical user interface. The program has several menus, including File, Edit, and Help. To represent the ID of each menu, you may decide to use a constant. A perfectly reasonable name for a constant referring to the Help menu ID is `kHelp`.

The name `kHelp` will work fine until one day you add a Help button to the main window. You also need a constant to refer to the ID of the button, but `kHelp` is already taken.

A possible solution for this is to put your constants in different namespaces, which are discussed in Chapter 1. You create two namespaces: `Menu` and `Button`. Each namespace has a `kHelp` constant and you use them as `Menu::kHelp` and `Button::kHelp`. Another and more preferred solution is to use enumerators, also introduced in Chapter 1.

Hungarian Notation

Hungarian Notation is a variable and data member naming convention that is popular with Microsoft Windows programmers. The basic idea is that instead of using single-letter prefixes such as `m`, you should use more verbose prefixes to indicate additional information. The following line of code displays the use of Hungarian Notation:

```
char* pszName; // psz means "pointer to a null-terminated string"
```

The term *Hungarian Notation* arose from the fact that its inventor, Charles Simonyi, is Hungarian. Some also say that it accurately reflects the fact that programs using Hungarian Notation end up looking as if they were written in a foreign language. For this latter reason, some programmers tend to dislike Hungarian Notation. In this book, prefixes are used, but not Hungarian Notation. Adequately named variables don't need much additional context information besides the prefix. For example, a data member named `mName` says it all.

NOTE *Good names convey information about their purpose without making the code unreadable.*

USING LANGUAGE FEATURES WITH STYLE

The C++ language lets you do all sorts of terribly unreadable things. Take a look at this wacky code:

```
i++ + ++i;
```

This is unreadable but more importantly, its behavior is undefined by the C++ standard. The problem is that `i++` uses the value of `i` but has a side effect of incrementing it. The standard does not say when this incrementing should be done, only that the side effect (increment) should be visible after the sequence point “`;`”, but the compiler can do it at any point in time during the execution of that line. It’s impossible to know which value of `i` will be used for the `++i` part. Running this code with different compilers and platforms can result in different values. The following is another example of code with undefined behavior which you should avoid.

```
a[i] = i++;
```

With all the power that the C++ language offers, it is important to consider how the language features can be used towards stylistic good instead of evil.

Use Constants

Bad code is often littered with “magic numbers.” In some function, the code might be using `2.71828`. Why `2.71828`? What does that value mean? People with a mathematical background might find it obvious that this represents an approximation of the transcendental value e , but most people don’t know this. The language offers constants to give a symbolic name to a value that doesn’t change, such as `2.71828`.

```
const double kApproximationForE = 2.71828182845904523536;
```

Use References Instead of Pointers

Traditionally, C++ programmers learn C first. In C, pointers were the only pass-by-reference mechanism, and they certainly worked just fine for many years. Pointers are still required in some cases, but in many situations you can switch to references. If you learned C first, you probably think that references don’t really add any new functionality to the language. You might think that they merely introduce a new syntax for functionality that pointers could already provide.

There are several advantages to using references rather than pointers. First, references are safer than pointers because they don’t deal directly with memory addresses and cannot be `nullptr`. Second, references are more stylistically pleasing than pointers because they use the same syntax as stack variables, avoiding symbols such as `*` and `&`. They’re also easy to use, so you should have no problem adopting references into your style palette. Unfortunately, some programmers think that if they

see an `&` in a function call, they know the called function is going to change the object and if they don't see the `&` it must be pass-by-value. With references, they say they don't know if the function is going to change the object unless they look at the function prototype. This is a wrong way of thinking. Passing in a pointer does not automatically mean that the object will be modified, because the parameter might be `const T*`. Both passing a pointer and a reference can modify the object or not depending on whether the function prototype uses `const T*`, `T*`, `const T&` or `T&`. So, you need to look at the prototype anyway to know if the function might change the object.

Another benefit of references is that they clarify ownership of memory. If you are writing a method and another programmer passes you a reference to an object, it is clear that you can read and possibly modify the object, but you have no easy way of freeing its memory. If you are passed a pointer, this is less clear. Do you need to delete the object to clean up memory? Or will the caller do that? The preferred way of handling memory is to use smart pointers introduced in Chapter 1.

Use Custom Exceptions

C++ makes it easy to ignore exceptions. Nothing about the language syntax forces you to deal with exceptions and you could easily write error-tolerant programs with traditional mechanisms such as returning `nullptr` or setting an error flag.

Exceptions provide a much richer mechanism for error handling, and custom exceptions allow you to tailor this mechanism to your needs. For example, a custom exception type for a web browser could include fields that specify the web page that contained the error, the network state when the error occurred, and additional context information.

Chapter 13 contains a wealth of information about exceptions in C++.

NOTE *Language features exist to help the programmer. Understand and make use of features that contribute to good programming style.*

FORMATTING

Many programming groups have been torn apart and friendships ruined over code-formatting arguments. In college, a friend of mine got into such a heated debate with a peer over the use of spaces in an `if` statement that people were stopping by to make sure that everything was okay.

If your organization has standards in place for code formatting, consider yourself lucky. You may not like the standards they have in place, but at least you won't have to argue about it. If everybody on your team is writing code their own way, try to be as tolerant as you can. As you'll see, some practices are just a matter of taste, while others actually make it difficult to work in teams.

The Curly Brace Alignment Debate

Perhaps the most frequently argued-about point is where to put the curly braces that demarcate a block of code. There are several styles of curly brace use. In this book, the curly brace is put on the same

line as the leading statement, except in the case of a function, class, or method name. This style is shown in the code that follows (and throughout the book).

```
void someFunction()
{
    if (condition()) {
        cout << "condition was true" << endl;
    } else {
        cout << "condition was false" << endl;
    }
}
```

This style conserves vertical space while still showing blocks of code by their indentation. Some programmers would argue that preservation of vertical space isn't relevant in real-world coding. A more verbose style is shown below.

```
void someFunction()
{
    if (condition())
    {
        cout << "condition was true" << endl;
    }
    else
    {
        cout << "condition was false" << endl;
    }
}
```

Some programmers are even liberal with use of horizontal space, yielding code like that in the following example.

```
void someFunction()
{
    if (condition())
    {
        cout << "condition was true" << endl;
    }
    else
    {
        cout << "condition was false" << endl;
    }
}
```

Of course, I won't recommend any particular style because I don't want hate mail.

NOTE *When selecting a style for denoting blocks of code, the important consideration is how well you can see which block falls under which condition simply by looking at the code.*

Coming to Blows over Spaces and Parentheses

The formatting of individual lines of code can also be a source of disagreement. Again, I won't advocate a particular approach, but a few styles are shown that you are likely to encounter.

In this book, I use a space after any keyword, a space before and after any operator, a space after every comma in a parameter list or a call, and parentheses to clarify the order of operations, as follows:

```
if (i == 2) {  
    j = i + (k / m);  
}
```

The alternative, shown next, treats `if` stylistically like a function, with no space between the keyword and the left parenthesis. Also, the parentheses used to clarify the order of operations inside of the `if` statement are omitted because they have no semantic relevance.

```
if( i == 2 ) {  
    j = i + k / m;  
}
```

The difference is subtle, and the determination of which is better is left to the reader, yet I can't move on from the issue without pointing out that `if` is not a function.

Spaces and Tabs

The use of spaces and tabs is not merely a stylistic preference. If your group does not agree on a convention for spaces and tabs, there are going to be major problems when programmers work jointly. The most obvious problem occurs when Alice uses four spaces to indent code and Bob uses five space tabs; neither will be able to display code properly when working on the same file. An even worse problem arises when Bob reformats the code to use tabs at the same time that Alice edits the same code; many source code control systems won't be able to merge in Alice's changes.

Most, but not all, editors have configurable settings for spaces and tabs. Some environments even adapt to the formatting of the code as it is read in, or always save using spaces even if the tab key is used for authoring. If you have a flexible environment, you have a better chance of being able to work with other people's code. Just remember that tabs and spaces are different because tabs can be any length and a space is always a space.

STYLISTIC CHALLENGES

Many programmers begin a new project by pledging that, this time, they will do everything right. Any time a variable or parameter shouldn't be changed, it'll be marked `const`. All variables will have clear, concise, readable names. Every developer will put the left curly brace on the subsequent line and will adopt the standard text editor and its conventions for tabs and spaces.

For a number of reasons, it is difficult to sustain this level of stylistic consistency. In the case of `const`, sometimes programmers just aren't educated about how to use it. You will eventually come across old code or a library function that isn't `const`-savvy. A good programmer will use `const_cast` to temporarily suspend the `const` property of a variable, but an inexperienced programmer will start to unwind the `const` property back from the calling function, once again ending up with a program that never uses `const`.

Other times, standardization of style comes up against programmers' own individual tastes and biases. Perhaps the culture of your team makes it impractical to enforce strict style guidelines. In such situations, you may have to decide which elements you really need to standardize (such as variable names and tabs) and which ones are safe to leave up to individuals (perhaps spacing and commenting style). You can even obtain or write scripts that will automatically correct style "bugs" or flag stylistic problems along with code errors. Some development environments, such as Microsoft Visual C++ 2013, support automatic formatting of code according to rules that you specify. This makes it trivial to write code that always follows the guidelines that have been configured.

SUMMARY

The C++ language provides a number of stylistic tools without any formal guidelines on how to use them. Ultimately, any style convention is measured by how widely it is adopted and how much it benefits the readability of the code. When coding as part of a team, you should raise issues of style early in the process as part of the discussion of what language and tools to use.

The most important point about style is to appreciate that it is an important aspect of programming. Teach yourself to check over the style of your code before you make it available to others. Recognize good style in the code you interact with and adopt the conventions that you and your organization find useful.

This chapter concludes the first part of this book. The next part discusses software design on a high level.

PART II

Professional C++ Software Design

- **CHAPTER 4:** Designing Professional C++ Programs
- **CHAPTER 5:** Designing with Objects
- **CHAPTER 6:** Designing for Reuse

4

Designing Professional C++ Programs

WHAT'S IN THIS CHAPTER?

- The definition of programming design
- The importance of programming design
- The aspects of design that are unique to C++
- The two fundamental themes for effective C++ design: abstraction and reuse
- The different types of code available for reuse
- The advantages and disadvantages of code reuse
- General strategies and guidelines for reusing code
- Open-source libraries
- The C++ standard library

Before writing a single line of code in your application, you should design your program. What data structures will you use? What classes will you write? This plan is especially important when you program in groups. Imagine sitting down to write a program with no idea what your coworker, who is working on the same program, is planning! In this chapter, you'll learn how to use the Professional C++ approach to C++ design.

Despite the importance of design, it is probably the most misunderstood and underused aspect of the software-engineering process. Too often programmers jump into applications without a clear plan: They design as they code. This approach can lead to convoluted and overly complicated designs. It also makes the development, debugging, and maintenance tasks more difficult. Although counterintuitive, investing extra time at the beginning of a project to design it properly actually saves time over the life of the project.

WHAT IS PROGRAMMING DESIGN?

Your *program design*, or *software design*, is the specification of the architecture that you will implement to fulfill the functional and performance requirements of the program. Informally, the design is how you plan to write the program. You should generally write your design in the form of a design document. Although every company or project has its own variation of a desired design document format, most design documents share the same general layout, including two main parts:

1. The gross subdivision of the program into subsystems, including interfaces and dependencies between the subsystems, data flow between the subsystems, input and output to and from each subsystem, and general threading model.
2. The details of each subsystem, including subdivision into classes, class hierarchies, data structures, algorithms, specific threading model, and error-handling specifics.

The design documents usually include diagrams and tables showing subsystem interactions and class hierarchies. UML, Unified Modeling Language, is the industry standard used for diagrams. Consult one of the references in Appendix B for details. The exact format of the design document is less important than the process of thinking about your design.

NOTE *The point of designing is to think about your program before you write it.*

You should generally try to make your design as good as possible before you begin coding. The design should provide a map of the program that any reasonable programmer could follow in order to implement the application. Of course, it is inevitable that the design will need to be modified once you begin coding and you encounter issues that you didn't think of earlier. Software-engineering processes have been designed to give you the flexibility to make these changes. The Spiral Model proposed by Barry W. Boehm is one example of such an iterative process whereby the application is developed according to cycles with each cycle containing at least a requirements analysis, design and an implementation part. Chapter 24 describes various software-engineering process models in more detail.

THE IMPORTANCE OF PROGRAMMING DESIGN

It's tempting to skip the analysis and design steps, or to perform it only cursorily, in order to begin programming as soon as possible. There's nothing like seeing code compiling and running to give you the impression that you have made progress. It seems like a waste of time to formalize a design, or to write down functional requirements when you already know, more or less, how you want to structure your program. Besides, writing a design document just isn't as much fun as coding. If you wanted to write papers all day, you wouldn't be a computer programmer! As a programmer myself, I understand this temptation to begin coding immediately, and have certainly succumbed to it on occasion. However, it will most likely lead to problems on all but the simplest projects. Whether or not you succeed without a design prior to the implementation depends on your experience as a programmer, your proficiency with commonly used design patterns, and how deep you understand C++, the problem domain and the requirements.

To help you understand the importance of programming design, imagine that you own a plot of land on which you want to build a house. When the builder shows up you ask to see the blueprints. “What blueprints?” he responds, “I know what I’m doing. I don’t need to plan every little detail ahead of time. Two-story house? No problem — I did a one-story house a few months ago — I’ll just start with that model and work from there.”

Suppose that you suspend your disbelief and allow the builder to proceed. A few months later you notice that the plumbing appears to run outside the house instead of inside the walls. When you query the builder about this anomaly he says, “Oh. Well, I forgot to leave space in the walls for the plumbing. I was so excited about this new drywall technology it just slipped my mind. But it works just as well outside, and functionality is the most important thing.” You’re starting to have your doubts about his approach, but, against your better judgment, you allow him to continue.

When you take your first tour of the completed building, you notice that the kitchen lacks a sink. The builder excuses himself by saying, “We were already two-thirds done with the kitchen by the time we realized there wasn’t space for the sink. Instead of starting over we just added a separate sink room next door. It works, right?”

Do the builder’s excuses sound familiar if you translate them to the software domain? Have you ever found yourself implementing an “ugly” solution to a problem like putting plumbing outside the house? For example, maybe you forgot to include locking in your queue data structure that is shared between multiple threads. By the time you realized the problem, you decided to just perform the locking manually on all places where the queue is used. Sure, it’s ugly, but it works, you said. That is, until someone new joins the project who assumes that the locking is built into the data structure, fails to ensure mutual exclusion in her access to the shared data, and causes a race condition bug that takes three weeks to track down. Note that this locking problem is just given as an example of an ugly workaround. Obviously, a professional C++ programmer would never decide to perform the locking manually on each queue access but would instead directly incorporate the locking inside the queue class, or make the queue class thread-safe in a lock-free manner.

Formalizing a design before you code helps you determine how everything fits together. Just as blueprints for a house show how the rooms relate to each other and work together to fulfill the requirements of the house, the design for a program shows how the subsystems of the program relate to each other and work together to fulfill the software requirements. Without a design plan, you are likely to miss connections between subsystems, possibilities for reuse or shared information, and the simplest ways to accomplish tasks. Without the “big picture” that the design gives, you might become so bogged down in individual implementation details that you lose track of the overarching architecture and goals. Furthermore, the design provides written documentation to which all members of the project can refer. If you use an iterative process like the Spiral Model mentioned earlier, you need to make sure to keep the design documentation up-to-date during each cycle of the process.

If the preceding analogy hasn’t convinced you to design before you code, here is an example where jumping directly into coding fails to lead to an optimal design. Suppose that you want to write a chess program. Instead of designing the entire program before you begin programming, you decide to jump in with the easiest parts and move slowly to the more difficult parts. Following the object-oriented perspective introduced in Chapter 1 and covered in more detail in Chapter 5, you decide to model your chess pieces with classes. You figure the pawn is the simplest chess piece, so you opt

to start there. After considering the features and behaviors of a pawn, you write a class with the properties and behaviors shown in the following table:

CLASS	PROPERTIES	BEHAVIORS
Pawn	Location on Board Color (Black or White) Captured	Move Check Move Legality Draw Promote (Upon Reaching Opposing Side of the Board)

Of course, you didn't actually write the table. You went straight to the implementation. Happy with that class you move on to the next easiest piece: the bishop. After considering its attributes and functionality, you write a class with the properties and behaviors shown in the next table:

CLASS	PROPERTIES	BEHAVIORS
Bishop	Location on Board Color (Black or White) Captured	Move Check Move Legality Draw

Again, you didn't generate a table, because you jumped straight to the coding phase. However, at this point you begin to suspect that you might be doing something wrong. The bishop and the pawn look similar. In fact, their properties are identical and they share many behaviors. Although the implementations of the move behavior might differ between the pawn and the bishop, both pieces need the ability to move. If you had designed your program before jumping into coding, you would have realized that the various pieces are actually quite similar, and that you should find some way to write the common functionality only once. Chapter 5 explains the object-oriented design techniques for doing that.

Furthermore, several aspects of the chess pieces depend on other subsystems of your program. For example, you cannot accurately represent the location on the board in a chess piece class without knowing how you will model the board. On the other hand, perhaps you will design your program so that the board manages pieces in a way that doesn't require them to know their own locations. In either case, encoding the location in the piece classes before designing the board leads to problems. To take another example, how can you write a draw method for a piece without first deciding your program's user interface? Will it be graphical or text-based? What will the board look like? The problem is that subsystems of a program do not exist in isolation — they interrelate with other subsystems. Most of the design work determines and defines these relationships.

DESIGNING FOR C++

There are several aspects of the C++ language that you need to keep in mind when designing for C++.

- C++ has an immense feature set. It is almost a complete superset of the C language, plus classes and objects, operator overloading, exceptions, templates, and many other features. The sheer size of the language makes design a daunting task.

- C++ is an object-oriented language. This means that your designs should include class hierarchies, class interfaces, and object interactions. This type of design is quite different from “traditional” design in C or other procedural languages. Chapter 5 focuses on object-oriented design in C++.
- C++ has numerous facilities for designing generic and reusable code. In addition to basic classes and inheritance, you can use other language facilities such as templates and operator overloading for effective design. Design techniques for reusable code are discussed in more details further in this chapter.
- C++ provides a useful standard library, including a string class, I/O facilities, and many common data structures and algorithms. All of these facilitate coding in C++.
- C++ is a language that readily accommodates many *design patterns*, or common ways to solve problems.

Tackling a design can be overwhelming. I have spent entire days scribbling design ideas on paper, crossing them out, writing more ideas, crossing those out, and repeating the process. Sometimes this process is helpful, and, at the end of those days (or weeks), leads to a clean, efficient design. Other times it can be frustrating, and leads nowhere, but it is not a waste of effort. You will most likely waste more time if you have to re-implement a design that turned out to be broken. It’s important to remain aware of whether or not you are making real progress. If you find that you are stuck, you can take one of the following actions:

- **Ask for help.** Consult a coworker, mentor, book, newsgroup, or web page.
- **Work on something else for a while.** Come back to this design choice later.
- **Make a decision and move on.** Even if it’s not an ideal solution, decide on something and try to work with it. An incorrect choice will soon become apparent. However, it may turn out to be an acceptable method. Perhaps there is no clean way to accomplish what you want to with this design. Sometimes you have to accept an “ugly” solution if it’s the only realistic strategy to fulfill your requirements. Whatever you decide, make sure you document your decision, so that you and others in the future know why you made it. This includes documenting designs that you have rejected and the rationale behind the rejection.

NOTE *Keep in mind that good design is hard, and getting it right takes practice. Don’t expect to become an expert overnight, and don’t be surprised if you find it more difficult to master C++ design than C++ coding.*

TWO RULES FOR C++ DESIGN

There are two fundamental design rules in C++: *abstraction* and *reuse*. These guidelines are so important that they can be considered themes of this book. They come up repeatedly throughout the text, and throughout effective C++ program designs in all domains.

Abstraction

The principle of *abstraction* is easiest to understand through a real-world analogy. A television is a simple piece of technology found in most homes. You are probably familiar with its features: You can turn it on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players. However, can you explain how it works inside the black box? That is, do you know how it receives signals over the air or through a cable, translates them, and displays them on the screen? Most people certainly can't explain how a television works, yet are quite capable of using it. That is because the television clearly separates its internal *implementation* from its external *interface*. We interact with the television through its interface: the power button, channel changer, and volume control. We don't know, nor do we care, how the television works; we don't care whether it uses a cathode ray tube or some sort of alien technology to generate the image on our screen. It doesn't matter because it doesn't affect the interface.

Benefiting from Abstraction

The abstraction principle is similar in software. You can use code without knowing the underlying implementation. As a trivial example, your program can make a call to the `sqrt()` function declared in the header file `<cmath>` without knowing what algorithm the function actually uses to calculate the square root. In fact, the underlying implementation of the square root calculation could change between releases of the library, and as long as the interface stays the same, your function call will still work. The principle of abstraction extends to classes as well. As introduced in Chapter 1, you can use the `cout` object of class `ostream` to stream data to standard output like this:

```
cout << "This call will display this line of text" << endl;
```

In this line, you use the documented interface of the `cout` insertion operator (`<<`) with a character array. However, you don't need to understand how `cout` manages to display that text on the user's screen. You only need to know the public interface. The underlying implementation of `cout` is free to change as long as the exposed behavior and interface remain the same.

Incorporating Abstraction in Your Design

You should design functions and classes so that you and other programmers can use them without knowing, or relying on, the underlying implementations. To see the difference between a design that exposes the implementation and one that hides it behind an interface, consider the chess program again. You might want to implement the chess board with a two-dimensional array of pointers to `ChessPiece` objects. You could declare and use the board like this:

```
ChessPiece* chessBoard[8][8];
...
ChessBoard[0][0] = new Rook();
```

However, that approach fails to use the concept of abstraction. Every programmer who uses the chess board knows that it is implemented as a two-dimensional array. Changing that implementation to something else, such as an array of vectors, would be difficult, because you would need to change every use of the board in the entire program. There is no separation of interface from implementation.

A better approach is to model the chess board as a class. You could then expose an interface that hides the underlying implementation details. Here is an example of the `ChessBoard` class:

```
class ChessBoard
{
public:
    // This example omits constructors, destructors, and assignment operator.
    void setPieceAt(ChessPiece* piece, int x, int y);
    ChessPiece& getPieceAt(int x, int y);
    bool isEmpty(int x, int y) const;
private:
    // This example omits data members.
};
```

Note that this interface makes almost no commitment to any underlying implementation. The `ChessBoard` could easily be a two-dimensional array, but the interface does not require it. Changing the implementation does not require changing the interface. Furthermore, the implementation can provide additional functionality, such as bounds checking.

There is only one small commitment to the underlying implementation because of the interface. The `getPieceAt()` function returns a reference, so it is recommended for the underlying implementation to not directly store objects in a collection, but pointers or better yet smart pointers to objects, to avoid bizarre aliasing problems, which can be hard to track down. For example, suppose a client of the `ChessBoard` class stores a reference received by a call to `getPieceAt()`. If the `ChessBoard` class directly stores objects in the underlying collection, this returned reference can become invalid when the `ChessBoard` class needs to reallocate memory for the collection. Storing pointers or smart pointers in the collection avoids this reference-validation problem.

This example, hopefully, has convinced you that abstraction is an important technique in C++ programming. Chapter 5 covers abstraction and object-oriented design in more detail, and Chapters 7 and 8 provide all the details about writing your own classes.

Reuse

The second fundamental rule of design in C++ is *reuse*. Again, it is helpful to examine a real-world analogy to understand this concept. Suppose that you give up your programming career in favor of work as a baker. On your first day of work, the head baker tells you to bake cookies. In order to fulfill his orders you find the recipe for chocolate-chip cookies, mix the ingredients, form cookies on the cookie sheet, and place the sheet in the oven. The head baker is pleased with the result.

Now, I'm going to point out something so obvious that it will surprise you: you didn't build your own oven in which to bake the cookies. Nor did you churn your own butter, mill your own flour, or form your own chocolate chips. I can hear you think, "That goes without saying." That's true if you're a real cook, but what if you're a programmer writing a baking simulation game? In that case, you would think nothing of writing every component of the program, from the chocolate chips to the oven. Or, you could save yourself time by looking around for code to reuse. Perhaps your office-mate wrote a cooking simulation game and has some nice oven code lying around. Maybe it doesn't do everything you need, but you might be able to modify it and add the necessary functionality.

Something else you took for granted is that you followed a recipe for the cookies instead of making up your own. Again, that goes without saying. However, in C++ programming, it does not go without saying. Although there are standard ways of approaching problems that arise over and over in C++, many programmers persist in reinventing these strategies in each design.

Reusing Code

The idea of using existing code is not new. You've been reusing code from the first day you printed something with `cout`. You didn't write the code to actually print your data to the screen. You used the existing `cout` implementation to do the work.

Unfortunately, programmers generally do not take advantage of available code. Your designs should take into account existing code and reuse it when appropriate. How to reuse existing code is discussed in depth later in this chapter.

Writing Reusable Code

The design theme of reuse applies to code you write as well as to code that you use. You should design your programs so that you can reuse your classes, algorithms, and data structures. You and your coworkers should be able to use these components in both the current project and in future projects. In general, you should avoid designing overly specific code that is applicable only to the case at hand.

One language technique for writing general-purpose code in C++ is the *template*. The following example shows a templated data structure. If you've never seen this syntax before, don't worry! Chapter 11 explains the syntax in depth.

Instead of writing a specific `ChessBoard` class that stores `ChessPieces`, as shown earlier, consider writing a generic `GameBoard` template that can be used for any type of two-dimensional board game such as chess or checkers. You would need only to change the class declaration so that it takes the piece to store as a template parameter instead of hard-coding it in the interface. The template could look something like this:

```
template <typename PieceType>
class GameBoard
{
public:
    // This example omits constructors, destructors, and assignment operator.
    void setPieceAt(PieceType* piece, int x, int y);
    PieceType& getPieceAt(int x, int y);
    bool isEmpty(int x, int y) const;
private:
    // This example omits data members.
};
```

With this simple change in the interface, you now have a generic game board class that you can use for any two-dimensional board game. Although the code change is simple, it is important to make these decisions in the design phase, so that you are able to implement the code effectively and efficiently.

Chapter 6 goes into more details on how to design your code with reuse in mind.

Reusing Ideas

As the baker example illustrates, it would be ludicrous to reinvent recipes for every dish that you make. However, programmers often make an equivalent mistake in their designs. Instead of using existing “recipes,” or *patterns*, for designing programs, they reinvent these techniques every time they design a program. However, many *design patterns* appear in myriad different C++ applications. As a C++ programmer, you should familiarize yourself with these patterns so that you can incorporate them effectively into your program designs.

For example, you might want to design your chess program so that you have a single `ErrorLogger` object that serializes all errors from different components to a log file. When you try to design your `ErrorLogger` class, you realize that it would be disastrous to have more than one object instantiated from the `ErrorLogger` class in a single program. You also want to be able to access this `ErrorLogger` object from anywhere in your program. These requirements of a single, globally accessible, instance of a class arise frequently in C++ programs, and there is a standard strategy to implement them, called the *singleton*. Thus, a good design at this point would specify that you want to use the singleton pattern. A detailed discussion of design patterns and techniques is outside the scope of this book. If you are interested, consult one of the references from Appendix B.

REUSING CODE

Experienced C++ programmers never start a project from scratch. They incorporate code from a wide variety of sources, such as the standard template library, open-source libraries, proprietary code bases in their workplace, and their own code from previous projects. In addition, good C++ programmers reuse approaches or strategies to address various common design issues. These strategies can range from a technique that worked for a past project to a formal *design pattern*. You should reuse code liberally in your designs. In order to make the most of this rule, you need to understand the types of code that you can reuse and the tradeoffs involved in code reuse. Note that reusing code does not mean copy-pasting existing code! In fact it means quite the opposite, reusing code without duplicating it.

A Note on Terminology

Before analyzing the advantages and disadvantages of code reuse, it is helpful to specify the terminology involved and to categorize the types of reused code. There are three categories of code available for reuse:

- Code you wrote yourself in the past
- Code written by a coworker
- Code written by a third party outside your current organization or company

There are also several ways that the code you use can be structured:

- **Stand-alone functions or classes.** When you reuse your own code or coworkers’ code, you will generally encounter this variety.

- **Libraries.** A *library* is a collection of code used to accomplish a specific task, such as parsing XML, or to handle a specific domain, such as cryptography. Other examples of functionality usually found in libraries include threads and synchronization support, networking, and graphics.
- **Frameworks.** A *framework* is a collection of code around which you design a program. For example, the Microsoft Foundation Classes (MFC) provide a framework for creating graphical user interface applications for Microsoft Windows. Frameworks usually dictate the structure of your program.

NOTE *A program uses a library but fits into a framework. Libraries provide specific functionality, while frameworks are fundamental to your program design and structure.*

Another term that arises frequently is *application programming interface*, or *API*. An API is an interface to a library or body of code for a specific purpose. For example, programmers often refer to the sockets API, meaning the exposed interface to the sockets networking library, instead of the library itself.

NOTE *Although people use the terms API and library interchangeably, they are not equivalent. The library refers to the implementation, while the API refers to the published interface to the library.*

For the sake of brevity, the rest of this chapter uses the term *library* to refer to any reused code, whether it is really a library, framework, or random collection of functions from your office-mate.

Deciding Whether or Not to Reuse Code

The rule to reuse code is easy to understand in the abstract. However, it's somewhat vague when it comes to the details. How do you know when it's appropriate to reuse code, and which code to reuse? There is always a tradeoff, and the decision depends on the specific situation. However, there are some general advantages and disadvantages to reusing code.

Advantages to Reusing Code

Reusing code can provide tremendous advantages to you and to your project.

- You may not know how to, or may not be able to justify the time to write the code you need. Would you really want to write code to handle formatted input and output? Of course not: That's why you use the standard C++ I/O streams.
- Your designs will be simpler because you will not need to design those components of the application that you reuse.

- The code that you reuse usually requires no debugging. You can often assume that library code is bug-free because it has already been tested and used extensively.
- Libraries handle more error conditions than would your first attempt at the code. You might forget obscure errors or edge cases at the beginning of the project, and would waste time fixing these problems later. Library code that you reuse has generally been tested extensively and used by many programmers before you, so you can assume that it handles most errors properly.
- Libraries generally are designed to be suspect of bad user inputs. Invalid requests, or requests not appropriate for the current state, usually result in suitable error notifications. For example, a request to seek to a nonexistent record in a database, or to read a record from a database which is not open, would have well-specified behavior from a library.
- Reusing code written by domain experts is safer than writing your own code for that area. For example, you should not attempt to write your own security code unless you are a security expert. If you need security or cryptography in your programs, use a library. Many seemingly minor details in code of that nature could compromise the security of the entire program if you got them wrong.
- Library code is constantly improving. If you reuse the code, you receive the benefits of these improvements without doing the work yourself. In fact, if the library writers properly separated the interface from the implementation, you can obtain these benefits by upgrading your library version without changing your interaction with the library. A good upgrade modifies the underlying implementation without changing the interface.

Disadvantages to Reusing Code

Unfortunately, there are also some disadvantages to reusing code.

- When you use only code that you wrote yourself, you understand exactly how it works. When you use libraries that you didn't write yourself, you must spend time understanding the interface and correct usage before you can jump in and use it. This extra time at the beginning of your project will slow your initial design and coding.
- When you write your own code, it does exactly what you want. Library code might not provide the exact functionality that you require.
- Even if the library code provides the exact functionality you need, it might not give you the performance that you desire. The performance might be bad in general, poor for your specific use case, or completely undocumented.
- Using library code introduces a Pandora's box of support issues. If you discover a bug in the library, what do you do? Often you don't have access to the source code, so you couldn't fix it even if you wanted to. If you have already invested significant time learning the library interface and using the library, you probably don't want to give it up, but you might find it difficult to convince the library developers to fix the bug on your time schedule. Also, if you are using a third-party library, what do you do if the library authors drop support for the library before you stop supporting the product that depends on it? Think carefully about this before you decide to use a library for which you cannot get source code.

- In addition to support problems, libraries present licensing issues which cover topics such as disclosure of your source, redistribution fees (often called binary license fees), credit attribution, and development licenses. You should carefully inspect the licensing issues before using any library. For example, some open-source libraries require you to make your own code open-source.
- Another consideration with reusing code is cross-platform portability. If you want to write a cross-platform application, make sure the libraries you use are also cross-platform portable.
- Reusing code requires a trust factor. You must trust whoever wrote the code by assuming that he or she did a good job. Some people like to have control over all aspects of their project, including every line of source code.
- Upgrading to a new version of the library can cause problems. The upgrade could introduce bugs which could have fatal consequences in your product. A performance related upgrade might optimize performance in certain cases but make it worse in your specific use-case.

Putting It Together to Make a Decision

Now that you are familiar with the terminology, advantages, and disadvantages of reusing code, you are better prepared to make the decision about whether or not to reuse code. Often, the decision is obvious. For example, if you want to write a graphical user interface (GUI) in C++ for Microsoft Windows, you should use a framework such as MFC or QT. You probably don't know how to write the underlying code to create a GUI in Windows, and more importantly, you don't want to waste the time to learn it. You will save person-years of effort by using a framework in this case.

However, other times the choice is less obvious. For example, if you are unfamiliar with a library or framework, and need only a simple data structure, it might not be worth the time to learn the entire framework to reuse only one component that you could write in a few days.

Ultimately, the decision is a choice that you need to make for your own particular needs. It often comes down to a tradeoff between the time it would take to write it yourself and the time required to find and learn how to use a library to solve the problem. Carefully consider how the advantages and disadvantages listed previously apply to your specific case, and decide which factors are most important to you. Finally, remember that you can always change your mind, which might even be not too much work if you handled the abstraction correctly.

Strategies for Reusing Code

When you use libraries, frameworks, coworkers' code, or your own code, there are several guidelines you should keep in mind.

Understand the Capabilities and Limitations

Take the time to familiarize yourself with the code. It is important to understand both its capabilities and its limitations. Start with the documentation and the published interfaces or APIs. Ideally, that will be sufficient to understand how to use the code. However, if the library doesn't provide a clear separation between interface and implementation, you may need to explore the

source code itself if provided. Also, talk to other programmers who have used the code and who might be able to explain its intricacies. You should begin by learning the basic functionality. If it's a library, what behaviors does it provide? If it's a framework, how does your code fit in? What classes should you derive from? What code do you need to write yourself? You should also consider specific issues depending on the type of code.

Here are some points to keep in mind:

- Is the code safe for multithreaded programs?
- Does the library impose any specific compiler settings on code using the library? If so, is that acceptable in your project?
- What initialization calls does the library or framework need? What cleanup does it need?
- On what other libraries does the library or framework depend?
- If you derive from a class, which constructor should you call on it? Which virtual methods should you override?
- If a call returns memory pointers, who is responsible for freeing the memory: the caller or the library? If the library is responsible, when is the memory freed? It's highly recommended to find out if you can use smart pointers to manage memory allocated by the library. Smart pointers are introduced in Chapter 1.
- What error conditions does the library call check for, and what does it assume? How does it handle errors? How does it notify the client program about errors? Avoid using libraries that pop up message boxes, issue messages to `stderr/cerr` or `stdout/cout`, or terminate the program.
- What are all the return values (by value or reference) from a call?
- What are all the possible exceptions thrown?

Understand the Performance

It is important to know the performance guarantees that the library or other code provides. Even if your particular program is not performance sensitive, you should make sure that the code you use doesn't have awful performance for your particular use.

Big-O Notation

Programmers generally discuss and document algorithm and library performance using *big-O notation*. This section explains the general concepts of algorithm complexity analysis and big-O notation without a lot of unnecessary mathematics. If you are already familiar with these concepts, you may skip this section.

Big-O notation specifies *relative*, rather than *absolute*, performance. For example, instead of saying that an algorithm runs in a specific amount of time, such as 300 milliseconds, big-O notation specifies how an algorithm performs as its input size increases. Examples of input sizes include the

number of items to be sorted by a sorting algorithm, the number of elements in a hash table during a key lookup, and the size of a file to be copied between disks.

NOTE Note that big-O notation applies only to algorithms whose speed depends on their inputs. It does not apply to algorithms that take no input or whose running time is random. In practice, you will find that the running times of most algorithms of interest depend on their input, so this limitation is not significant.

To be more formal: Big-O notation specifies algorithm run time as a function of its input size, also known as the *complexity* of the algorithm. However, that's not as complicated as it sounds. For example, an algorithm with a performance that is linear as a function of its input size takes twice as long to process twice as many elements. Thus, if it takes 2 seconds to process 500 elements, it will take 4 seconds to process 1000 elements. That is, you could graph the performance versus input size as a straight line. Big-O notation summarizes the algorithm performance like this: $O(n)$. The O just means that you're using big-O notation, while the n represents the input size. $O(n)$ specifies that the algorithm speed is a direct linear function of the input size.

Unfortunately, not all algorithms have performance that is linear with respect to the input size. Computer programs would run a lot faster if that were true. The following table summarizes the common complexities, in order of their performance from best to worst:

ALGORITHM COMPLEXITY	BIG-O NOTATION	EXPLANATION	EXAMPLE ALGORITHMS
Constant	$O(1)$	Running time is independent of input size.	Accessing a single element in an array
Logarithmic	$O(\log n)$	The running time is a function of the logarithm base 2 of the input size.	Finding an element in a sorted list using binary search
Linear	$O(n)$	The running time is directly proportional to the input size.	Finding an element in an unsorted list
Linear Logarithmic	$O(n \log n)$	The running time is a function of the linear times the logarithmic function of the input size.	Merge sort
Quadratic	$O(n^2)$	The running time is a function of the square of the input size.	A slower sorting algorithm like selection sort
Exponential	$O(2^n)$	The running time is an exponential function of the input size.	Optimized traveling salesman problem

There are two advantages to specifying performance as a function of the input size instead of in absolute numbers:

1. It is platform independent. Specifying that a piece of code runs in 200 milliseconds on one computer says nothing about its speed on a second computer. It is also difficult to compare two different algorithms without running them on the same computer with the exact same load. On the other hand, performance specified as a function of the input size is applicable to any platform.
2. Performance as a function of input size covers all possible inputs to the algorithm with one specification. The specific time in seconds that an algorithm takes to run covers only one specific input, and says nothing about any other input.

Sometimes the statistical expectations are taken into account when working with big-O notation in which case big-O represents the *expected time*. For example, a linear search is often said to be $O(n/2)$ because statistically about half of the elements need to be searched each time. The number of cases that are found in less than $O(n/2)$ are compensated for by the number of cases that require more than $O(n/2)$ time.

Tips for Understanding Performance

Now that you are familiar with big-O notation, you are prepared to understand most performance documentation. The C++ standard template library in particular describes its algorithm and data structure performance using big-O notation. However, big-O notation is sometimes insufficient or even misleading. Consider the following issues whenever you think about big-O performance specifications:

- If an algorithm takes twice as long to work on twice as much data, it doesn't say anything about how long it took in the first place! If the algorithm is written badly but scales well, it's still not something you want to use. For example, suppose the algorithm makes unnecessary disk accesses. That probably wouldn't affect the big-O time but would be very bad for overall performance.
- Along those lines, it's difficult to compare two algorithms with the same big-O running time. For example, if two different sorting algorithms both claim to be $O(n \log n)$, it's hard to tell which is really faster without running your own tests.
- The big-O notation describes the time complexity of an algorithm asymptotically, as the input size grows to infinity. For small inputs, big-O time can be very misleading. An $O(n^2)$ algorithm might actually perform better than an $O(\log n)$ algorithm on small input sizes. Consider your likely input sizes before making a decision.

In addition to considering big-O characteristics, you should look at other facets of the algorithm performance. Here are some guidelines to keep in mind:

- You should consider how often you intend to use a particular piece of library code. Some people find the "90/10" rule helpful: 90 percent of the running time of most programs is spent in only 10 percent of the code (Hennessy and Patterson, "Computer Architecture, A Quantitative Approach, Fourth Edition", 2006, Morgan Kaufmann, 2002). If the library code you intend to use falls in the oft-exercised 10 percent category of your code, you should make sure to analyze its performance characteristics carefully. On the other hand, if it falls

into the oft-ignored 90 percent of the code, you should not spend much time analyzing its performance because it will not benefit your overall program performance very much.

- Don't trust the documentation. Always run performance tests to determine if library code provides acceptable performance characteristics.

Understand Platform Limitations

Before you start using library code, make sure that you understand on which platforms it runs. That might sound obvious, but even libraries that claim to be cross-platform might contain subtle differences on different platforms.

Also, platforms include not only different operating systems but different versions of the same operating system. If you write an application that should run on Solaris 8, Solaris 9, and Solaris 10, ensure that any libraries you use also support all those releases. You cannot assume either forward or backward compatibility across operating system versions. That is, just because a library runs on Solaris 9 doesn't mean that it will run on Solaris 10 and vice versa.

Understand Licensing and Support

Using third-party libraries often introduces complicated licensing issues. You must sometimes pay license fees to third-party vendors for the use of their libraries. There may also be other licensing restrictions, including export restrictions. Additionally, open-source libraries are sometimes distributed under licenses that require any code that links with them to be open source as well. A number of licenses commonly used by open-source libraries are discussed later in the chapter.

WARNING *Make sure that you understand the license restrictions of any third-party libraries you use if you plan to distribute or sell the code you develop. When in doubt, consult a legal expert.*

Using third-party libraries also introduces support issues. Before you use a library, make sure that you understand the process for submitting bugs, and that you realize how long it will take for bugs to be fixed. If possible, determine how long the library will continue to be supported so that you can plan accordingly.

Interestingly, even using libraries from within your own organization can introduce support issues. You may find it just as difficult to convince a coworker in another part of your company to fix a bug in his or her library as you would to convince a stranger in another company to do the equivalent. In fact, you may even find it harder, because you're not a paying customer. Make sure that you understand the politics and organizational issues within your own organization before using internal libraries.

Know Where to Find Help

Using libraries and frameworks can sometimes be daunting at first. Fortunately, there are many avenues of support available. First of all, consult the documentation that accompanies the library. If the library is widely used, such as the standard template library (STL), or the MFC, you should be

able to find a good book on the topic. In fact, for help with the STL, consult Chapters 15 and later of this book. If you have specific questions not addressed by books and product documentation, try searching the web. Type your question in your search engine of choice to find web pages that discuss the library. For example, when you search for the phrase “introduction to C++ STL” you will find several hundred websites about C++ and the STL. Also, many websites contain their own private newsgroups or forums on specific topics for which you can register.

WARNING *A note of caution: Don't believe everything you read on the web! Web pages do not necessarily undergo the same review process as printed books and documentation, and may contain inaccuracies.*

Prototype

When you first sit down with a new library or framework, it is often a good idea to write a quick prototype. Trying out the code is the best way to familiarize yourself with the library’s capabilities. You should consider experimenting with the library even before you tackle your program design so that you are intimately familiar with the library’s capabilities and limitations. This empirical testing will allow you to determine the performance characteristics of the library as well.

Even if your prototype application looks nothing like your final application, time spent prototyping is not a waste. Don’t feel compelled to write a prototype of your actual application. Write a dummy program that just tests the library capabilities you want to use. The point is only to familiarize yourself with the library.

WARNING *Due to time constraints, programmers sometimes find their prototypes morphing into the final product. If you have hacked together a prototype that is insufficient as the basis for the final product, make sure that it doesn't get used that way.*

Bundling Third-Party Applications

Your project might include multiple applications. Perhaps you need a web server front end to support your new e-commerce infrastructure. It is possible to bundle third-party applications, such as a web server, with your software. This approach takes code reuse to the extreme in that you reuse entire applications. However, most of the caveats and guidelines for using libraries apply to bundling third-party applications as well. Specifically, make sure that you understand the legality and licensing ramifications of your decision.

NOTE *Consult a legal expert whose specialty is Intellectual Property before bundling third-party applications with your software distributions.*

Also, the support issue becomes more complex. If customers encounter a problem with your bundled web server, should they contact you or the web server vendor? Make sure that you resolve this issue *before* you release the software.

Open-Source Libraries

Open-source libraries are an increasingly popular class of reusable code. The general meaning of *open-source* is that the source code is available for anyone to look at. There are formal definitions and legal rules about including source with all your distributions, but the important thing to remember about open-source software is that anyone (including you) can look at the source code. Note that open-source applies to more than just libraries. In fact, the most famous open-source product is probably the Linux operating system. Google Chrome and Mozilla Firefox are two examples of famous open-source web browsers.

The Open-Source Movements

Unfortunately, there is some confusion in terminology in the open-source community. First of all, there are two competing names for the movement (some would say two separate, but similar, movements). Richard Stallman and the GNU project use the term *free software*. Note that the term free does not imply that the finished product must be available without cost. Developers are welcome to charge as much or as little as they want. Instead, the term free refers to the freedom for people to examine the source code, modify the source code, and redistribute the software. Think of the free in free speech rather than the free in free beer. You can read more about Richard Stallman and the GNU project at www.gnu.org.

The Open Source Initiative uses the term *open-source software* to describe software in which the source must be available. As with free software, open-source software does not require the product or library to be available without cost. However, an important difference with free software is that open-source software is not required to give you the freedom to use, modify, and redistribute it. You can read more about the Open Source Initiative at www.opensource.org.

There are several licensing options available for open-source projects. One of them is the GNU Public License (GPL). However, using a library under the GPL requires you to make your own product open-source as well. On the other hand, an open-source project can use a licensing option like Boost, OpenBSD, CodeGuru, Boost Software License, BSD, Code Project Open License, CodeProject, Creative Commons License, and so on, which allow using the open-source library in a closed-source product.

Because the name “open-source” is less ambiguous than “free software,” this book uses “open-source” to refer to products and libraries with which the source code is available. The choice of name is not intended to imply endorsement of the open-source philosophy over the free software philosophy: It is only for ease of comprehension.

Finding and Using Open-Source Libraries

Regardless of the terminology, you can gain amazing benefits from using open-source software. The main benefit is functionality. There are a plethora of open-source C++ libraries available for varied tasks: from XML parsing to cross-platform error logging.

Although open-source libraries are not required to provide free distribution and licensing, many open-source libraries are available without monetary cost. You will generally be able to save money in licensing fees by using open-source libraries.

Finally, you are often but not always free to modify open-source libraries to suit your exact needs.

Most open-source libraries are available on the web. For example, searching for “open-source C++ library XML parsing” results in a list of links to XML libraries in C and C++. There are also a few open-source portals where you can start your search, including:

- www.boost.org
- www.gnu.org
- www.sourceforge.net

Guidelines for Using Open-Source Code

Open-source libraries present several unique issues and require new strategies. First of all, open-source libraries are usually written by people in their “free” time. The source base is generally available for any programmer who wants to pitch in and contribute to development or bug fixing. As a good programming citizen, you should try to contribute to open-source projects if you find yourself reaping the benefits of open-source libraries. If you work for a company, you may find resistance to this idea from your management because it does not lead directly to revenue for your company. However, you might be able to convince management that indirect benefits, such as exposure of your company name, and perceived support from your company for the open-source movement, should allow you to pursue this activity.

Second, because of the distributed nature of their development, and lack of single ownership, open-source libraries often present support issues. If you desperately need a bug fixed in a library, it is often more efficient to make the fix yourself than to wait for someone else to do it. If you do fix bugs, you should make sure to put the fixes into the public source base for the library. Even if you don’t fix any bugs, make sure to report problems that you find so that other programmers don’t waste time encountering the same issues.

The C++ Standard Library

The most important library that you will use as a C++ programmer is the C++ standard library. As its name implies, this library is part of the C++ standard, so any standards-conforming compiler should include it. The standard library is not monolithic: It includes several disparate components, some of which you have been using already. You may even have assumed they were part of the core language. Chapters 15 and later go into more details about the standard library.

C Standard Library

Because C++ is mostly a superset of C, most of the C library is still available. Its functionality includes mathematical functions such as `abs()`, `sqrt()`, and `pow()`, and error-handling helpers such as `assert()` and `errno`. Additionally, the C library facilities for manipulating character arrays as strings, such as `strlen()` and `strcpy()`, and the C-style I/O functions, such as `printf()` and `scanf()`, are all available in C++.

NOTE C++ provides better strings and I/O support than C. Even though the C-style strings and I/O routines are available in C++, you should avoid them in favor of C++ strings (Chapter 2) and I/O streams (Chapter 12).

Note that the C header files have different names in C++. These names should be used instead of the C library names, because they are less likely to result in name conflicts. For details of the C libraries, consult a Standard Library Reference, for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>.

Deciding Whether or Not to Use the STL

The STL was designed with functionality, performance, and orthogonality as its priorities. The benefits of using it are substantial. Imagine having to track down pointer errors in linked list or balanced binary tree implementations, or debug a sorting algorithm that isn't sorting properly. If you use the STL correctly, you will rarely, if ever, need to perform that kind of coding. Chapters 15 and later provide in-depth information on the STL functionality.

DESIGNING WITH PATTERNS AND TECHNIQUES

Learning the C++ language and becoming a good C++ programmer are two very different things. If you sat down and read the C++ standard, memorizing every fact, you would know C++ as well as anybody else. However, until you gain some experience by looking at code and writing your own programs, you wouldn't necessarily be a good programmer. The reason is that the C++ syntax defines what the language can do in its raw form, but doesn't say anything about how each feature should be used.

As they become more experienced in using the C++ language, C++ programmers develop their own individual ways of using the features of the language. The C++ community at large has also built some standard ways of leveraging the language, some formal and some informal. Throughout this book, I point out these reusable applications of the language, known as *design techniques* and *design patterns*. Some patterns and techniques will seem obvious to you because they are simply a formalization of the obvious solution. Others describe novel solutions to problems you've encountered in the past. Some present entirely new ways of thinking about your program organization.

It is important for you to familiarize yourself with these patterns and techniques so that you can recognize when a particular design problem calls for one of these solutions. There are many more techniques and patterns applicable to C++ than those described in this book. A detailed discussion of design techniques and design patterns is outside the scope of this book. If you are interested in more details, consult a reference from Appendix B.

DESIGNING A CHESS PROGRAM

This section introduces a systematic approach to designing a C++ program in the context of a simple chess game application. In order to provide a complete example, some of the steps refer to concepts covered in later chapters. You should read this example now, in order to obtain an overview of the design process, but you might also consider rereading it after you have finished later chapters.

Requirements

Before embarking on the design, it is important to possess clear requirements for the program's functionality and efficiency. Ideally, these requirements would be documented in the form of a requirements specification. The requirements for the chess program would contain the following types of specifications, although in more detail and number:

- The program will support the standard rules of chess.
- The program will support two human players. The program will not provide an artificially intelligent computer player.
- The program will provide a text-based interface:
 - The program will render the game board and pieces in plain text.
 - Players will express their moves by entering numbers representing locations on the chessboard.

The requirements ensure that you design your program so that it performs as its users expect.

Design Steps

You should take a systematic approach to designing your program, working from the general to the specific. The following steps do not always apply to all programs, but they provide a general guideline. Your design should include diagrams and tables as appropriate. An industry standard for making diagrams is called UML (Unified Modeling Language). UML defines a multitude of standard diagrams you can use for documenting software designs, for example class diagrams, sequence diagrams, and so on. I recommend using UML or at least UML-like diagrams where applicable. However, I'm no advocate of strictly adhering to the UML syntax because having a clear, understandable diagram is more important than a syntactically correct one.

Divide the Program into Subsystems

Your first step is to divide your program into its general functional subsystems and to specify the interfaces and interactions between the subsystems. At this point, you should not worry about specifics of data structures and algorithms, or even classes. You are trying only to obtain a general feel for the various parts of the program and their interactions. You can list the subsystems in a table that expresses the high-level behaviors or functionality of the subsystem, the interfaces exported from the subsystem to other subsystems, and the interfaces consumed, or used, by this subsystem on other subsystems. The recommended design for this chess game is to have a clear separation between storing the data and displaying the data by using the *Model-View-Controller* (MVC) paradigm. This paradigm models the notion that many applications commonly deal with a set of data, one or

more views on that data, and manipulation of the data. In MVC, a set of data is called the *model*, a *view* is a particular visualization of the model, and the *controller* is the piece of code that changes the model in response to some event. The three components of MVC interact in a feedback loop; Actions are handled by the controller, which adjusts the model, resulting in a change to the view(s). Using this paradigm, you can easily switch between having a text-based interface and a graphical user interface. A table for the chess game subsystems could look like this:

Subsystem Name	Instances	Functionality	Interfaces Exported	Interfaces Consumed
GamePlay	1	Starts game Controls game flow Controls drawing Declares winner Ends game	Game Over	Take Turn (on Player) Draw (on ChessBoardView)
ChessBoard	1	Stores chess pieces Checks for ties and checkmates	Get Piece At Set Piece At	Game Over (on GamePlay)
ChessBoardView	1	Draws the associated ChessBoard	Draw	Draw (on ChessPieceView)
ChessPiece	32	Moves itself Checks for legal moves	Move Check Move	Get Piece At (on ChessBoard) Set Piece At (on ChessBoard)
ChessPieceView	32	Draws the associated ChessPiece	Draw	None
Player	2	Interacts with user: prompts user for move, obtains user's move Moves pieces	Take Turn	Get Piece At (on ChessBoard) Move (on ChessPiece) Check Move (on ChessPiece)
ErrorLogger	1	Writes error messages to log file	Log Error	None

As this table shows, the functional subsystems of this chess game include a GamePlay subsystem, a ChessBoard and ChessBoardView, 32 ChessPieces and ChessPieceViews, two Players, and one ErrorLogger. However, that is not the only reasonable approach. In software design, as in programming itself, there are often many different ways to accomplish the same goal. Not all solutions are equal: Some are certainly better than others. However, there are often several equally valid methods.

A good division into subsystems separates the program into its basic functional parts. For example, a Player is a subsystem distinct from the ChessBoard, ChessPieces, or GamePlay. It wouldn't make sense to lump the players into the GamePlay subsystem because they are logically separate subsystems. Other choices might not be as obvious.

In this MVC design, the ChessBoard and ChessPiece subsystems are part of the Model. The ChessBoardView and ChessPieceView are part of the View and the Player is part of the Controller.

Because it is often difficult to visualize subsystem relationships from tables, it is usually helpful to show the subsystems of a program in a diagram where arrows represent calls from one subsystem to another.

Choose Threading Models

It's too early in the design phase to think about how to multithread specific loops in algorithms you will write. However, in this step, you choose the number of high-level threads in your program and specify their interactions. Examples of high-level threads are a UI thread, an audio-playing thread, a network communication thread, and so on. In multithreaded designs, you should try to avoid shared data as much as possible because it will make your designs simpler and safer. If you cannot avoid shared data, you should specify locking requirements. If you are unfamiliar with multithreaded programs, or your platform does not support multithreading, then you should make your programs single-threaded. However, if your program has several distinct tasks, each of which could work in parallel, it might be a good candidate for multiple threads. For example, graphical user interface applications often have one thread performing the main application work and another thread waiting for the user to press buttons or select menu items. Multithreaded programming is covered in Chapter 23.

The chess program needs only one thread to control the game flow.

Specify Class Hierarchies for Each Subsystem

In this step, you determine the class hierarchies that you intend to write in your program. The chess program needs a class hierarchy to represent the chess pieces. The hierarchy could work as shown in Figure 4-1.

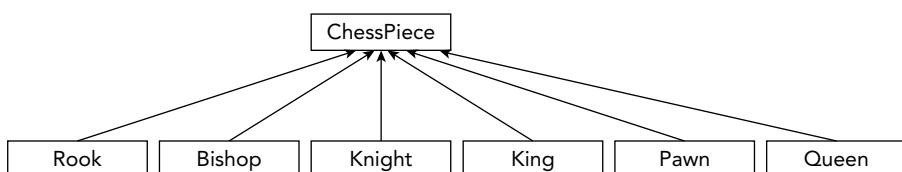


FIGURE 4-1

In this hierarchy, a generic `ChessPiece` class serves as the abstract base class. A similar hierarchy is required for the `ChessPieceView` class.

Another class hierarchy can be used for the `ChessBoardView` class to make it possible to have a text-based interface or a graphical user interface for the game. Figure 4-2 shows an example hierarchy that allows the chess board to be displayed as text on a console or as a 2D or 3D rendering.

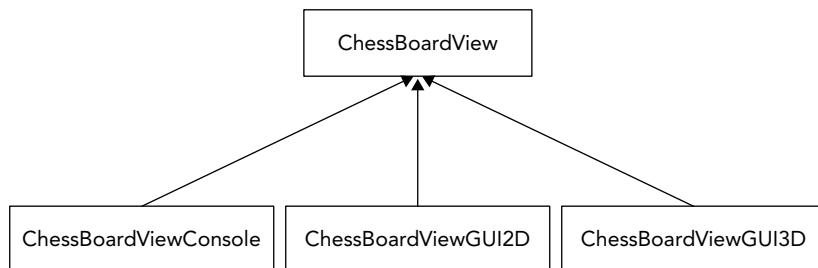


FIGURE 4-2

A similar hierarchy is required for the Player controller and for the individual classes of the `ChessPieceView` hierarchy.

Chapter 5 explains the details of designing classes and class hierarchies.

Specify Classes, Data Structures, Algorithms, and Patterns for Each Subsystem

In this step, you consider a greater level of detail, and specify the particulars of each subsystem, including the specific classes that you write for each subsystem. It may well turn out that you model each subsystem itself as a class. This information can again be summarized in a table:

Subsystem	Classes	Data Structures	Algorithms	Patterns
GamePlay	GamePlay class	GamePlay object includes one <code>ChessBoard</code> object and two <code>Player</code> objects.	Gives each player a turn to play	None
ChessBoard	ChessBoard class	ChessBoard object stores a two-dimensional representation of 32 <code>ChessPieces</code> .	Checks for win or tie after each move	None

Subsystem	Classes	Data Structures	Algorithms	Patterns
ChessBoardView	ChessBoardView abstract base class Concrete derived classes ChessBoardViewConsole, ChessBoardViewGUI2D ...	Stores information on how to draw a chess board	Draws a chess board	Observer
ChessPiece	ChessPiece abstract base class Rook, Bishop, Knight, King, Pawn, and Queen derived classes	Each piece stores its location on the chess board.	Piece checks for legal move by querying chess board for pieces at various locations.	None
ChessPieceView	ChessPieceView abstract base class Derived classes RookView, BishopView ... and concrete derived classes RookViewConsole, RookViewGUI2D ...	Stores information on how to draw a chess piece	Draws a chess piece	Observer
Player	Player abstract base class Concrete derived classes PlayerConsole, PlayerGUI2D, ...	None	Prompt user for move, check if move is legal, and move piece.	Mediator
ErrorLogger	One ErrorLogger class	A queue of messages to log	Buffers messages and writes them to a log file	Singleton pattern to ensure only one ErrorLogger object

This section of the design document would normally present the actual interfaces for each class, but this example will forgo that level of detail.

Designing classes and choosing data structures, algorithms, and patterns can be tricky. You should always keep in mind the rules of abstraction and reuse discussed earlier in this chapter. For abstraction, the key is to consider the interface and the implementation separately. First, specify the interface from the perspective of the user. Decide *what* you want the component to do. Then decide

how the component will do it by choosing data structures and algorithms. For reuse, familiarize yourself with standard data structures, algorithms, and patterns. Also, make sure you are aware of the standard library code in C++, as well as any proprietary code available in your workplace.

Specify Error Handling for Each Subsystem

In this design step, you delineate the error handling in each subsystem. The error handling should include both system errors, such as memory allocation failures, and user errors, such as invalid entries. You should specify whether each subsystem uses exceptions. You can again summarize this information in a table:

SUBSYSTEM	HANDLING SYSTEM ERRORS	HANDLING USER ERRORS
GamePlay	Logs an error with the ErrorLogger, shows a message to the user and gracefully shuts down the program if unable to allocate memory for ChessBoard or Players	Not applicable (no direct user interface)
ChessBoard ChessPiece	Logs an error with the ErrorLogger and throws an exception if unable to allocate memory	Not applicable (no direct user interface)
ChessBoardView ChessPieceView	Logs an error with the ErrorLogger and throws an exception if something goes wrong during rendering	Not applicable (no direct user interface)
Player	Logs an error with the ErrorLogger and throws an exception if unable to allocate memory	Sanity-checks user move entry to ensure that it is not off the board; prompts user for another entry. Checks each move legality before moving the piece; if illegal, prompts user for another move.
ErrorLogger	Attempts to log an error, informs user, and gracefully shuts down the program if unable to allocate memory	Not applicable (no direct user interface)

The general rule for error handling is to handle everything. Think hard about all possible error conditions. If you forget one possibility, it will show up as a bug in your program! Don't treat anything as an "unexpected" error. Expect all possibilities: memory allocation failures, invalid user entries, disk failures, and network failures, to name a few. However, as the table for the chess game shows, you should handle user errors differently from internal errors. For example, a user entering an invalid move should not cause your chess program to terminate.

Chapter 13 discusses error handling in more depth.

SUMMARY

In this chapter, you learned about the professional C++ approach to design. Hopefully it convinced you that software design is an important first step in any programming project. You also learned about some of the aspects of C++ that make design difficult, including its object-oriented focus, its large feature set and standard library, and its facilities for writing generic code. With this information, you are better prepared to tackle C++ design.

This chapter introduced two design themes. The concept of abstraction, or separating interface from implementation, permeates this book and should be a guideline for all your design work.

The notion of reuse, both of code and ideas, also arises frequently in real-world projects, and in this text. You learned that your C++ designs should include both reuse of code, in the form of libraries and frameworks, and reuse of ideas, in the form of techniques and patterns. You should write your code to be as reusable as possible. Also remember about the tradeoffs and about specific guidelines for reusing code, including understanding the capabilities and limitations, the performance, licensing and support models, the platform limitations, prototyping, and where to find help. You also learned about performance analysis and big-O notation. Now that you understand the importance of design and the basic design themes, you are ready for the rest of Part II. Chapter 5 describes strategies for using the object-oriented aspects of C++ in your design.

5

Designing with Objects

WHAT'S IN THIS CHAPTER?

- What object-oriented programming design is
- How you can define relationships between different objects
- The importance of abstraction and how to use it in your designs

Now that you have developed an appreciation for good software design from Chapter 4, it's time to pair the notion of objects with the concept of good design. The difference between programmers who use objects in their code and those who truly grasp object-oriented programming comes down to the way their objects relate to each other and to the overall design of the program.

This chapter begins with a very brief description of procedural programming (C-style), followed by a detailed discussion of object-oriented programming (OOP). Even if you've been using objects for years, you will want to read this chapter for some new ideas regarding how to think about objects. A discussion of the different kinds of relationships between objects includes pitfalls programmers often succumb to when building an object-oriented program. You will also learn how the principal of abstraction relates to objects.

When thinking about procedural programming or object-oriented programming, the most important point to remember is that they just represent different ways of reasoning about what's going on in your program. Too often, programmers get bogged down in the syntax and jargon of OOP before they adequately understand what an object is. This chapter is light on code and heavy on concepts and ideas. For specifics on C++ object syntax, see Chapters 7, 8, and 9.

AM I THINKING PROCEDURALLY?

A procedural language, such as C, divides code into small pieces each of which (ideally) accomplishes a single task. Without procedures in C, all your code would be lumped together inside `main()`. Your code would be difficult to read, and your coworkers would be annoyed, to say the least.

The computer doesn't care if all your code is in `main()` or if it's split into bite-sized pieces with descriptive names and comments. Procedures are an abstraction that exists to help you, the programmer, as well as those who read and maintain your code. The concept is built around a fundamental question about your program — *What does this program do?* By answering that question in English, you are thinking procedurally. For example, you might begin designing a stock selection program by answering as follows: First, the program obtains stock quotes from the Internet. Then, it sorts this data by specific metrics. Next, it performs analysis on the sorted data. Finally, it outputs a list of buy and sell recommendations. When you start coding, you might directly turn this mental model into C functions: `retrieveQuotes()`, `sortQuotes()`, `analyzeQuotes()`, and `outputRecommendations()`.

NOTE Even though C refers to procedures as “functions,” C is not a functional language. The term functional is very different from procedural and refers to languages like Lisp, which use an entirely different abstraction.

The procedural approach tends to work well when your program follows a specific list of steps. In large modern applications, however, there is rarely a linear sequence of events. Often a user is able to perform any command at any time. Procedural thinking also says nothing about data representation. In the previous example, there was no discussion of what a stock quote actually is.

If the procedural mode of thought sounds like the way you approach a program, don't worry. Once you realize that OOP is simply an alternative, more flexible, way of thinking about software, it'll come naturally.

THE OBJECT-ORIENTED PHILOSOPHY

Unlike the procedural approach, which is based on the question *What does this program do?*, the object-oriented approach asks another question: *What real-world objects am I modeling?* OOP is based on the notion that you should divide your program not into tasks, but into models of physical objects. While this seems abstract at first, it becomes clearer when you consider physical objects in terms of their *classes, components, properties, and behaviors*.

Classes

A class helps distinguish an object from its definition. Consider the orange. There's a difference between talking about oranges in general as tasty fruit that grows on trees and talking about a specific orange, such as the one that's currently dripping juice on my keyboard.

When answering the question *What are oranges?* you are talking about the *class* of things known as oranges. All oranges are fruit. All oranges grow on trees. All oranges are some shade of orange. All oranges have some particular flavor. A class is simply the encapsulation of what defines a classification of objects.

When describing a specific orange, you are talking about an object. All objects belong to a particular class. Because the object on my desk is an orange, I know that it belongs to the orange class. Thus, I know that it is a fruit that grows on trees. I can further say that it is a medium shade of orange and ranks “mighty tasty” in flavor. An object is an *instance* of a class — a particular item with characteristics that distinguish it from other instances of the same class.

As a more concrete example, reconsider the stock selection application from above. In OOP, “stock quote” is a class because it defines the abstract notion of what makes up a quote. A specific quote, such as “current Microsoft stock quote,” would be an object because it is a particular instance of the class.

From a C background, think of classes and objects as analogous to types and variables. In fact, in Chapter 7, you’ll see that the syntax for classes is similar to the syntax for C structs.

Components

If you consider a complex real-world object, such as an airplane, it should be fairly easy to see that it is made up of smaller *components*. There’s the fuselage, the controls, the landing gear, the engines, and numerous other parts. The ability to think of objects in terms of their smaller components is essential to OOP, just as the breaking up of complicated tasks into smaller procedures is fundamental to procedural programming.

A component is essentially the same thing as a class, just smaller and more specific. A good object-oriented program might have an `Airplane` class, but this class would be huge if it fully described an airplane. Instead, the `Airplane` class deals with many smaller, more manageable, components. Each of these components might have further subcomponents. For example, the landing gear is a component of an airplane, and the wheel is a component of the landing gear.

Properties

Properties are what distinguish one object from another. Going back to the `Orange` class, recall that all oranges are defined as having some shade of orange and a particular flavor. These two characteristics are properties. All oranges have the same properties, just with different values. My orange has a “mighty tasty” flavor, but yours may have a “terribly unpleasant” flavor.

You can also think about properties on the class level. As recognized earlier, all oranges are fruit and grow on trees. These are properties of the fruit class whereas the specific shade of orange is determined by the particular fruit object. Class properties are shared by all objects of a class, while object properties are present in all objects of the class, but with different values.

In the stock selection example, a stock quote has several object properties, including the name of the company, its ticker symbol, the current price, and other statistics.

Properties are the characteristics that describe an object. They answer the question *What makes this object different?*

Behaviors

Behaviors answer either of two questions: *What does this object do?* Or, *What can I do to this object?* In the case of an orange, it doesn't do a whole lot, but we can do things to it. One behavior is that it can be eaten. Like properties, you can think of behaviors on the class level or the object level. All oranges can pretty much be eaten in the same way. However, they might differ in some other behavior, such as being rolled down an incline, where the behavior of a perfectly round orange would differ from that of a more oblate one.

The stock selection example provides some more practical behaviors. As you recall, when thinking procedurally, we determined that our program needs to analyze stock quotes as one of its functions. Thinking in OOP, you might decide that a stock quote object can analyze itself. Analysis becomes a behavior of the stock quote object.

In object-oriented programming, the bulk of functional code is moved out of procedures and into classes. By building classes that have certain behaviors and defining how they interact, OOP offers a much richer mechanism for attaching code to the data on which it operates.

Bringing It All Together

With these concepts, you could take another look at the stock selection program and redesign it in an object-oriented manner.

As discussed, “stock quote” would be a fine class to start with. To obtain the list of quotes, the program needs the notion of a group of stock quotes, which is often called a *collection*. So a better design might be to have a class that represents a “collection of stock quotes,” which is made up of smaller components that represent a single “stock quote.”

Moving on to properties, the collection class would have at least one property — the actual list of quotes received. It might also have additional properties, such as the exact date and time of the most recent retrieval and the number of quotes obtained. As for behaviors, the “collection of stock quotes” would be able to talk to a server to get the quotes and provide a sorted list of quotes. This is the “retrieve quotes” behavior.

The stock quote class would have the properties discussed earlier — name, symbol, current price, and so on. Also, it would have an analyze behavior. You might consider other behaviors, such as buying and selling the stock.

It is often useful to jot down diagrams showing the relationship between components. Figure 5-1 uses multiple lines to indicate that one “collection of stock quotes” contains many “stock quote” objects.

Another useful way of visualizing classes is to list properties and behaviors when brainstorming the object representation of a program, as in the following table:

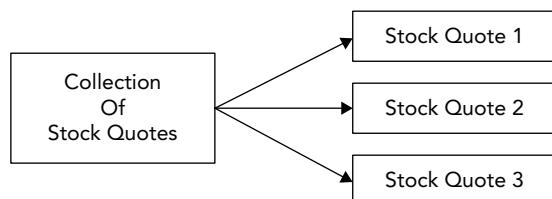


FIGURE 5-1

CLASS	ASSOCIATED COMPONENTS	PROPERTIES	BEHAVIORS
Orange	Seeds Juice Pulp	Color Flavor	Eat Roll Toss Peel Squeeze
Collection of Stock Quotes	Made up of individual Stock Quote objects	Individual Quotes Timestamp Number of Quotes	Retrieve quotes Sort quotes by various criteria
Stock Quote	None (yet)	Company Name Ticker Symbol Current Price, and so on	Analyze Buy shares Sell shares

LIVING IN A WORLD OF OBJECTS

When programmers make the transition from thinking procedurally to the object-oriented paradigm, they often experience an epiphany about the combination of properties and behaviors into objects. Some programmers find themselves revisiting the design of programs they're working on and rewriting certain pieces as objects. Others might be tempted to throw all the code away and restart the project as a fully object-oriented application.

There are two major approaches to developing software with objects. To some people, objects simply represent a nice encapsulation of data and functionality. These programmers sprinkle objects throughout their programs to make the code more readable and easier to maintain. Programmers taking this approach slice out isolated pieces of code and replace them with objects like a surgeon implanting a pacemaker. There is nothing inherently wrong with this approach. These people see objects as a tool that is beneficial in many situations. Certain parts of a program just "feel like an object," like the stock quote. These are the parts that can be isolated and described in real-world terms.

Other programmers adopt the OOP paradigm fully and turn everything into an object. In their minds, some objects correspond to real-world things, such as an orange or a stock quote, while others encapsulate more abstract concepts, such as a sorter or an undo object. The ideal approach is probably somewhere in between these extremes. Your first object-oriented program might really have been a traditional procedural program with a few objects sprinkled in. Or perhaps you went whole hog and made everything an object, from a class representing an `int` to a class representing the main application. Over time, you will find a happy medium.

Overobjectification

There is often a fine line between designing a creative object-oriented system and annoying everybody else on your team by turning every little thing into an object. As Freud used to say, sometimes a variable is just a variable. Okay, that's a paraphrase of what he said.

Perhaps you’re designing the next bestselling Tic-Tac-Toe game. You’re going all-out OOP on this one, so you sit down with a cup of coffee and a notepad to sketch out your classes and objects. In games like this, there’s often an object that oversees game play and is able to detect the winner. To represent the game board, you might envision a `Grid` object that will keep track of the markers and their locations. In fact, a component of the grid could be the `Piece` object that represents an X or an O.

Wait, back up! This design proposes to have a class that represents an X or an O. That is perhaps object overkill. After all, can’t a `char` represent an X or an O just as well? Better yet, why can’t the `Grid` just use a two-dimensional array of an enumerated type? Does a `Piece` object just complicate the code? Take a look at the following table representing the proposed piece class:

CLASS	ASSOCIATED COMPONENTS	PROPERTIES	BEHAVIORS
<code>Piece</code>	None	X or O	None

The table is a bit sparse, strongly hinting that what we have here may be too granular to be a full-fledged object.

On the other hand, a forward-thinking programmer might argue that while `Piece` is a pretty meager class as it currently stands, making it into an object allows future expansion without any real penalty. Perhaps down the road, this will be a graphical application and it might be useful to have the `Piece` class support drawing behavior. Additional properties could be the color of the `Piece` or whether the `Piece` was the most recently moved.

Another solution might be to think about the *state* of a grid square instead of using pieces. The state of a square can be `Empty`, `X` or `O`. To make the design future-proof to support a graphical application, you could design an abstract base class `State` with concrete derived classes `StateEmpty`, `StateX` and `StateO` which know how to render themselves.

Obviously, there is no right answer. The important point is that these are issues that you should consider when designing your application. Remember that objects exist to help programmers manage their code. If objects are being used for no reason other than to make the code “more object-oriented,” something is wrong.

Overly General Objects

Perhaps a worse annoyance than objects that shouldn’t be objects is objects that are too general. All OOP students start with examples like “orange” — things that are objects, no question about it. In real life coding, objects can get pretty abstract. Many OOP programs have an “application object,” despite the fact that an application isn’t really something you can envision in material form. Yet it may be useful to represent the application as an object because the application itself has certain properties and behaviors.

An overly general object is an object that doesn’t represent a particular thing at all. The programmer may be attempting to make an object that is flexible or reusable, but ends up with one that is confusing. For example, imagine a program that organizes and displays media. It can catalog your photos, organize your digital music collection, and serve as a personal journal. The overly

general approach is to think of all these things as “media” objects and build a single class that can accommodate all of the formats. It might have a property called “data” that contains the raw bits of the image, song, or journal entry, depending on the type of media. It might have a behavior called “perform” that appropriately draws the image, plays the song, or brings up the journal entry for editing.

The clues that this class is too general are in the names of the properties and behaviors. The word “data” has little meaning by itself — you have to use a general term here because this class has been overextended to three very different uses. Similarly, “perform” will do very different things in the three different cases. Finally, this design is too general because “media” isn’t a particular object, not in the user interface, not in real life, and not even in the programmer’s mind. A major clue that a class is too general is when many ideas in the programmers mind all unite as a single object, as shown in Figure 5-2.

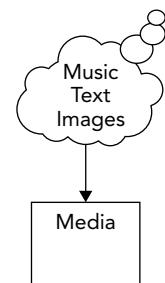


FIGURE 5-2

OBJECT RELATIONSHIPS

As a programmer, you will certainly encounter cases where different classes have characteristics in common, or at least seem somehow related to each other. For example, although creating a “media” object to represent images, music, and text in a digital catalog program is too general, these objects do share characteristics. You may want all of them to keep track of the date and time that they were last modified, or you might want them all to support a delete behavior.

Object-oriented languages provide a number of mechanisms for dealing with such relationships between objects. The tricky part is to understand what the relationship actually is. There are two main types of object relationships — a *has-a* relationship and an *is-a* relationship.

The Has-A Relationship

Objects engaged in a has-a, or *aggregation*, relationship follow the pattern A has a B, or A contains a B. In this type of relationship, you can envision one object as part of another. Components, as defined earlier, generally represent a has-a relationship because they describe objects that are made up of other objects.

A real-world example of this might be the relationship between a zoo and a monkey. You could say that a zoo has a monkey or a zoo contains a monkey. A simulation of a zoo in code would have a zoo object, which has a monkey component.

Often, thinking about user interface scenarios is helpful in understanding object relationships. This is so because even though not all UIs are implemented in OOP (though these days, most are), the visual elements on the screen translate well into objects. One UI analogy for a has-a relationship is a window that contains a button. The button and the window are clearly two separate objects but they are obviously related in some way. Since the button is inside the window, we say that the window has a button.

Figure 5-3 shows various real-world and user interface has-a relationships.

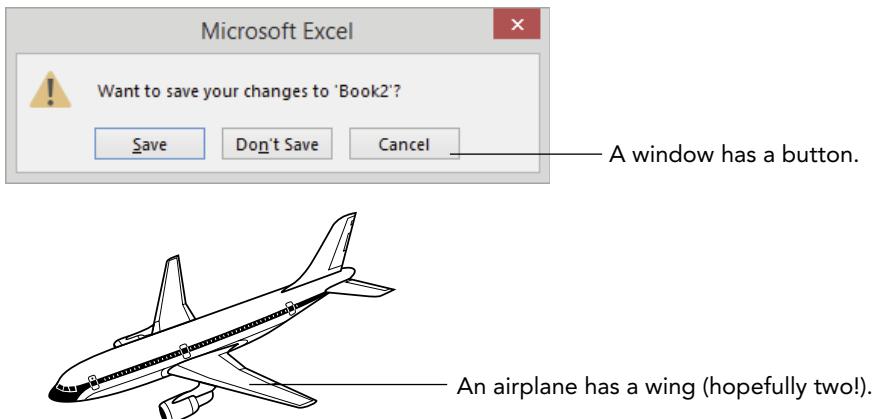


FIGURE 5-3

The Is-A Relationship (Inheritance)

The is-a relationship is such a fundamental concept of object-oriented programming that it has many names, including *deriving*, *subclassing*, *extending*, and *inheriting*. Classes model the fact that the real world contains objects with properties and behaviors. Inheritance models the fact that these objects tend to be organized in hierarchies. These hierarchies indicate is-a relationships.

Fundamentally, inheritance follows the pattern A is a B or A is really quite a bit like B — it can get tricky. To stick with the simple case, revisit the zoo, but assume that there are other animals besides monkeys. That statement alone has already constructed the relationship — a monkey is an animal. Similarly, a giraffe is an animal, a kangaroo is an animal, and a penguin is an animal. So what? Well, the magic of inheritance comes when you realize that monkeys, giraffes, kangaroos, and penguins have certain things in common. These commonalities are characteristics of animals in general.

What this means for the programmer is that you can define an `Animal` class that encapsulates all of the properties (size, location, diet, etc.) and behaviors (move, eat, sleep) that pertain to every animal. The specific animals, such as monkeys, become derived classes of `Animal` because a monkey contains all the characteristics of an animal. Remember, a monkey is an animal plus some additional characteristics that make it distinct. Figure 5-4 shows an inheritance diagram for animals. The arrows indicate the direction of the is-a relationship.

Just as monkeys and giraffes are different types of animals, a user interface often has different types of buttons. A checkbox, for example, is a button. Assuming that a button is simply a UI element that

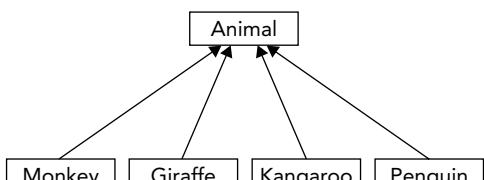


FIGURE 5-4

can be clicked and performs an action, a `Checkbox` extends the `Button` class by adding state — whether the box is checked or unchecked.

When relating classes in an is-a relationship, one goal is to factor common functionality into the *base class*, the class that other classes extend. If you find that all of your derived classes have code that is similar or exactly the same, consider how you could move some or all of the code into the base class. That way, any changes that need to be made only happen in one place and future derived classes get the shared functionality “for free.”

Inheritance Techniques

The preceding examples cover a few of the techniques used in inheritance without formalizing them. When deriving classes, there are several ways that the programmer can distinguish a class from its *parent class* or *base class* or *superclass*. A derived class may use one or more of these techniques and they are recognized by completing the sentence *A is a B that . . .*

Adding Functionality

A derived class can augment its parent by adding additional functionality. For example, a `monkey` is an animal that can swing from trees. In addition to having all of the behaviors of `Animal`, the `Monkey` class also has a `swing from trees` behavior, which is specific to only the `Monkey` class.

Replacing Functionality

A derived class can replace or *override* a behavior of its parent entirely. For example, most animals move by walking, so you might give the `Animal` class a `move` behavior that simulates walking. If that’s the case, a `kangaroo` is an animal that moves by hopping instead of walking. All the other properties and behaviors of the `Animal` base class still apply, but the `Kangaroo` derived class simply changes the way that the `move` behavior works. Of course, if you find yourself replacing *all* of the functionality of your base class, it may be an indication that inheriting was not the correct thing to do after all, unless the base class is an *abstract* base class. An abstract base class forces each of the derived classes to implement a certain behavior. You cannot create instances of an abstract base class. Abstract classes are discussed in Chapter 9.

Adding Properties

A derived class can also add new properties to the ones that were inherited from the base class. A `penguin` has all the properties of an animal but also has a `beak size` property.

Replacing Properties

C++ provides a way of overriding properties similar to the way you can override behaviors. However, doing so is rarely appropriate, because it hides the property from the base class, i.e. the base class can have a specific value for a property with a certain name, while the derived class can have another value for the property with the same name. Hiding is explained in more details in Chapter 9. It’s important not to get the notion of replacing a property confused with the notion of derived classes having different values for properties. For example, all animals have a `diet` property that indicates what they eat. Monkeys eat bananas and penguins eat `fish`, but neither of these is replacing the `diet` property — they simply differ in the value assigned to the property.

Polymorphism versus Code Reuse

Polymorphism is the notion that objects that adhere to a standard set of properties and behaviors can be used interchangeably. A class definition is like a contract between objects and the code that interacts with them. By definition, any `Monkey` object must support the properties and behaviors of the `Monkey` class.

This notion extends to base classes as well. Because all monkeys are animals, all `Monkey` objects support the properties and behaviors of the `Animal` class as well.

Polymorphism is a beautiful part of object-oriented programming because it truly takes advantage of what inheritance offers. In a zoo simulation, you could programmatically loop through all of the animals in the zoo and have each animal move once. Since all animals are members of the `Animal` class, they all know how to move. Some of the animals have overridden the move behavior, but that's the best part — our code simply tells each animal to move without knowing or caring what type of animal it is. Each one moves whichever way it knows how.

There is another reason to use inheritance besides polymorphism. Often, it's just a matter of leveraging existing code. For example, if you need a class that plays music with an echo effect, and your coworker has already written one that plays music without any effects, you might be able to extend the existing class and add in the new functionality. The *is-a* relationship still applies (an echo music player is a music player that adds an echo effect), but you didn't intend for these classes to be used interchangeably. What you end up with are two separate classes, used in completely different parts of the programs (or maybe even in different programs entirely) that happen to be related only to avoid reinventing the wheel.

The Fine Line between Has-A and Is-A

In the real world, it's pretty easy to classify has-a and is-a relationships between objects. Nobody would claim that an orange has a fruit — an orange *is a* fruit. In code, things sometimes aren't so clear.

Consider a hypothetical class that represents a hash table. A hash table is a data structure that efficiently maps a key to a value. For example, an insurance company could use a `Hashtable` class to map member IDs to names so that given an ID, it's easy to find the corresponding member name. The member ID is the *key* and the member name is the *value*.

In a standard hash table implementation, every key has a single value. If the ID 14534 maps to the member name “Kleper, Scott”, it cannot also map to the member name “Kleper, Marni”. In most implementations, if you tried to add a second value for a key that already has a value, the first value would go away. In other words, if the ID 14534 mapped to “Kleper, Scott” and you then assigned the ID 14534 to “Kleper, Marni”, then Scott would effectively be uninsured, as shown in the following sequence, which shows two calls to a hypothetical hash table `insert()` behavior and the resulting contents of the hash table. The notation `hash.insert` jumps ahead a bit to C++ object syntax. Just think of it as saying “use the `insert` behavior of the `hash` object.”

```
hash.insert(14534, "Kleper, Scott");
```

KEYS	VALUES
14534	“Kleper, Scott” [string]

```
hash.insert(14534, "Kleper, Marni");
```

KEYS	VALUES
14534	"Kleper, Marni" [string]

It's not difficult to imagine uses for a data structure that's *like* a hash table, but allows multiple values for a given key. In the insurance example, a family might have several names that correspond to the same ID. Because such a data structure is very similar to a hash table, it would be nice to leverage that functionality somehow. A hash table can have only a single value as a key, but that value can be anything. Instead of a string, the value could be a collection (such as an array or a list) containing the multiple values for the key. Every time you add a new member for an existing ID, add the name to the collection. This would work as shown in the following sequence.

```
Collection collection;           // Make a new collection.
collection.insert("Kleper, Scott"); // Add a new element to the collection.
hash.insert(14534, collection);   // Insert the collection into the table.
```

KEYS	VALUES
14534	{"Kleper, Scott"}

```
Collection collection = hash.get(14534); // Retrieve the existing collection.
collection.insert("Kleper, Marni");      // Add a new element to the collection.
hash.insert(14534, collection);          // Replace the collection with the updated one.
```

KEYS	VALUES
14534	{"Kleper, Scott", "Kleper, Marni"} [collection]

Messing around with a collection instead of a string is tedious and requires a lot of repetitive code. It would be preferable to wrap up this multiple-value functionality in a separate class, perhaps called a `MultiHash`. The `MultiHash` class would work just like `Hashtable` except that behind the scenes, it would store each value as a collection of strings instead of a single string. Clearly, `MultiHash` is somehow related to `Hashtable` because it is still using a hash table to store the data. What is unclear is whether that constitutes an *is-a* or a *has-a* relationship.

To start with the *is-a* relationship, imagine that `MultiHash` is a derived class of `Hashtable`. It would have to override the behavior that adds an entry into the table so that it would either create a collection and add the new element, or retrieve the existing collection and add the new element. It would also override the behavior that retrieves a value. It could, for example, append all the values for a given key together into one string. This seems like a perfectly reasonable design. Even though it overrides all the behaviors of the base class, it will still make use of the base class's behaviors by using the original behaviors within the derived class. This approach is shown in Figure 5-5.

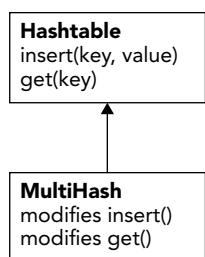


FIGURE 5-5

Now consider it as a has-a relationship.

MultiHash is its own class, but it *contains* a Hashtable object. It probably has an interface very similar to Hashtable, but it need not be the same. Behind the scenes, when a user adds something to the MultiHash, it is really wrapped in a collection and put in a Hashtable object. This also seems perfectly reasonable and is shown in Figure 5-6.

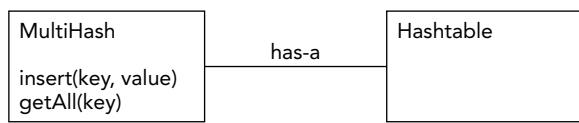


FIGURE 5-6

So, which solution is right? There's no clear answer, though a friend of mine who has written a MultiHash class for production use, viewed it as a has-a relationship. The main reason was to allow modifications to the exposed interface without worrying about maintaining hash table functionality. For example, in Figure 5-6, the get behavior was changed to `getAll`, making it clear that this would get all the values for a particular key in a MultiHash. Additionally, with a has-a relationship, you don't have to worry about any hash table functionality bleeding through. For example, if the hash table class supported a behavior that would get the total number of values, it would report the number of collections unless MultiHash knew to override it.

That said, one could make a convincing argument that a MultiHash actually is a Hashtable with some new functionality, and it should have been an is-a relationship. The point is that there is sometimes a fine line between the two relationships, and you will need to consider how the class is going to be used and whether what you are building just leverages some functionality from another class or really is that class with modified or new functionality.

The following table represents the arguments for and against taking either approach for the MultiHash class.

	IS-A	HAS-A
Reasons For	<ul style="list-style-type: none"> Fundamentally, it's the same abstraction with different characteristics. It provides (almost) the same behaviors as Hashtable. 	<ul style="list-style-type: none"> MultiHash can have whatever behaviors are useful without needing to worry about what behaviors Hashtable has. The implementation could change to something other than a Hashtable without changing the exposed behaviors.
Reasons Against	<ul style="list-style-type: none"> A hash table by definition has one value per key. To say MultiHash is a hash table is blasphemy! MultiHash overrides both behaviors of Hashtable, a strong sign that something about the design is wrong. Unknown or inappropriate properties or behaviors of Hashtable could "bleed through" to MultiHash. 	<ul style="list-style-type: none"> In a sense, MultiHash reinvents the wheel by coming up with new behaviors. Some additional properties and behaviors of Hashtable might have been useful.

The reasons against using an is-a relationship in this case are pretty strong. In fact, after years of experience, I recommend to opt for a has-a relationship over an is-a relationship if you have the choice.

Note that the `Hashtable` and `MultiHash` are used here to demonstrate the difference between the is-a and has-a relationships. In your own code, it is recommended to use one of the standard hash table classes instead of writing your own. The C++ standard library provides an `unordered_map` class, which you should use instead of the `Hashtable` and an `unordered_multimap` class, which you should use instead of the `MultiHash`. Both of these standard classes are discussed in Chapter 16.

The Not-A Relationship

As you consider what type of relationship classes have, consider whether or not they actually have a relationship at all. Don't let your zeal for object-oriented design turn into a lot of needless class/derived class relationships.

One pitfall occurs when things are obviously related in the real world but have no actual relationship in code. OO hierarchies need to model *functional* relationships, not artificial ones. Figure 5-7 shows relationships that are meaningful as ontologies or hierarchies, but are unlikely to represent a meaningful relationship in code.

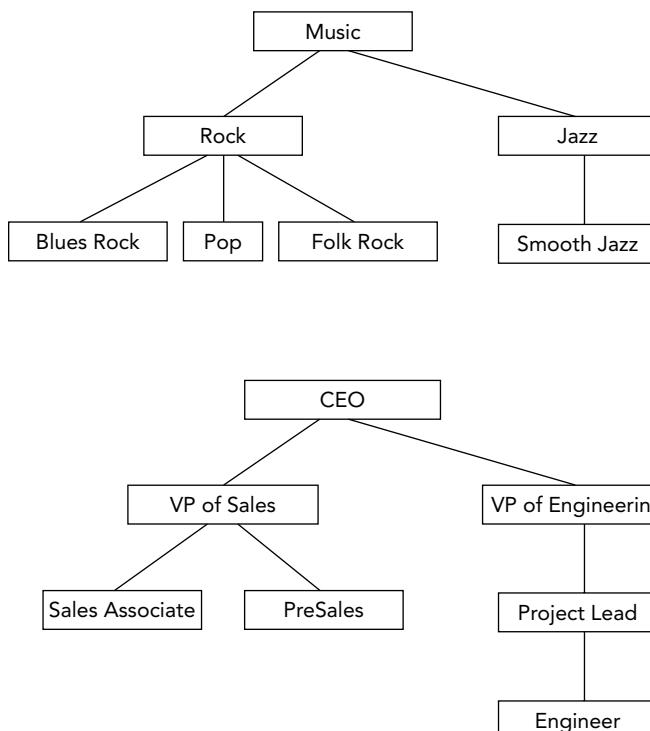


FIGURE 5-7

The best way to avoid needless inheritance is to sketch out your design first. For every class and derived class, write down what properties and behaviors you’re planning on putting there. You should rethink your design if you find that a class has no particular properties or behaviors of its own, or if all of those properties and behaviors are completely overridden by its derived classes, except when working with abstract base classes as mentioned earlier.

Hierarchies

Just as a class A can be a base class of B, B can also be a base class of C. Object-oriented hierarchies can model multilevel relationships like this. A zoo simulation with more animals might be designed with every animal as a derived class of a common `Animal` class as shown in Figure 5-8.

As you code each of these derived classes, you might find that a lot of them are similar. When this occurs, you should consider putting in a common parent. Realizing that `Lion` and `Panther` both move the same way and have the same diet might indicate a possible `BigCat` class. You could further subdivide the `Animal` class to include `WaterAnimal`, and `Marsupial`. A more hierarchical design that leverages this commonality is shown in Figure 5-9.

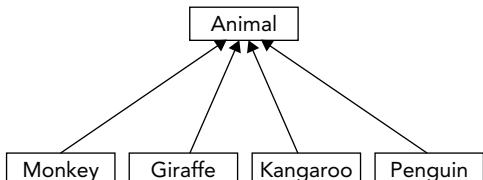


FIGURE 5-8

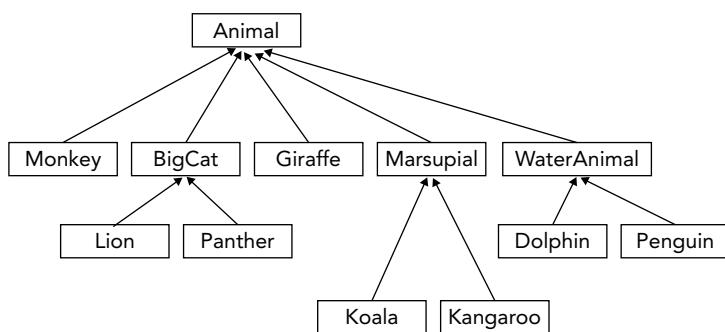


FIGURE 5-9

A biologist looking at this hierarchy may be disappointed — a penguin isn’t really in the same family as a dolphin. However, it underlines a good point — in code, you need to balance real-world relationships with shared functionality relationships. Even though two things might be very closely related in the real world, they might have a not-a relationship in code because they really don’t share functionality. You could just as easily divide animals into mammals and fish, but that wouldn’t factor any commonality to the base class.

Another important point is that there could be other ways of organizing the hierarchy. The preceding design is organized mostly by how the animals move. If it were instead organized by the animals’ diet or height, the hierarchy could be very different. In the end, what matters is how the classes will be used. The needs will dictate the design of the object hierarchy.

A good object-oriented hierarchy accomplishes the following:

- Organizes classes into meaningful functional relationships
- Supports code reuse by factoring common functionality to base classes
- Avoids having derived classes that override much of the parent's functionality, unless the parent is an abstract class.

Multiple Inheritance

Every example so far has had a single inheritance chain. In other words, a given class has, at most, one immediate parent class. This does not have to be the case. Through multiple inheritance, a class can have more than one base class.

Figure 5-10 shows a multiple inheritance design. There is still a base class called `Animal`, which is further divided by size. A separate hierarchy categorizes by diet, and a third takes care of movement. Each type of animal is then a derived class of all three of these classes, as shown by different lines.

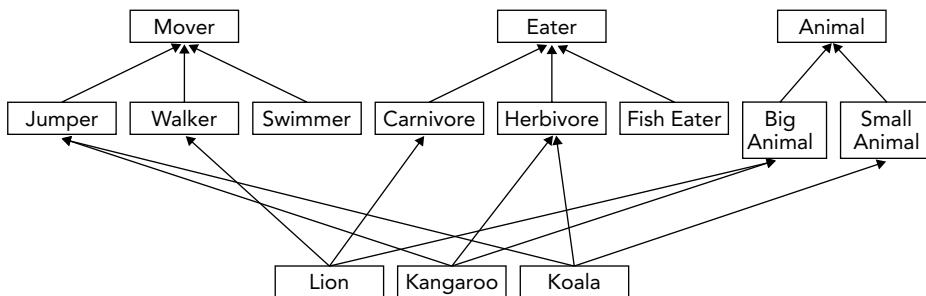


FIGURE 5-10

In a user interface context, imagine an image that the user can click on. This object seems to be both a button and an image so the implementation might involve inheriting from both the `Image` class and the `Button` class, as shown in Figure 5-11.

Multiple inheritance can be very useful in certain cases, but it also has a number of disadvantages that you should always keep in mind. Many programmers dislike multiple inheritance. C++ has explicit support for such relationships, though the Java language does away with them altogether, except for inheriting from multiple interfaces (abstract base classes). There are several reasons to which multiple inheritance critics point.

First, visualizing multiple inheritance is complicated. As you saw in Figure 5-10, even a simple class diagram can become very complicated when there are multiple hierarchies and crossing lines. Class hierarchies are supposed to make it easier for the programmer to understand the relationships between code. With multiple inheritance, a class could have several parents that are in no way related to each other. With so many classes contributing code to your object, can you really keep track of what's going on?

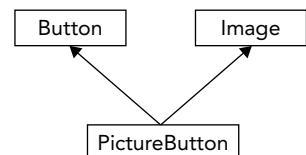


FIGURE 5-11

Second, multiple inheritance can destroy otherwise clean hierarchies. In the animal example, switching to a multiple inheritance approach means that the `Animal` base class is less meaningful because the code that describes animals is now separated into three separate hierarchies. While the design illustrated in Figure 5-10 shows three clean hierarchies, it's not difficult to imagine how they could get messy. For example, what if you realize that not only do all `Jumpers` move in the same way, they also eat the same things? Because there are separate hierarchies, there is no way to join the concepts of movement and diet without adding yet another derived class.

Third, implementation of multiple inheritance is complicated. What if two of your base classes implement the same behavior in different ways? Can you have two base classes that are themselves a derived class of a common base class? These possibilities complicate the implementation because structuring such intricate relationships in code is difficult both for the author and a reader.

The reason that other languages can leave out multiple inheritance is that it is usually avoidable. By rethinking your hierarchy you can often avoid introducing multiple inheritance when you have control over the design of a project.

Mixin Classes

Mixin classes represent another type of relationship between classes. In C++, a mixin class is implemented syntactically just like multiple inheritance, but the semantics are refreshingly different. A mixin class answers the question “What *else* is this class able to do?” and the answer often ends with “-able.” Mixin classes are a way that you can add functionality to a class without committing to a full is-a relationship. You can think of it as a *shares-with* relationship.

Going back to the zoo example, you might want to introduce the notion that some animals are “pettable.” That is, there are some animals that visitors to the zoo can pet, presumably without being bitten or mauled. You might want all pettable animals to support the behavior “be pet.” Since pettable animals don’t have anything else in common and you don’t want to break the existing hierarchy you’ve designed, `Pettable` makes a great mixin class.

Mixin classes are used frequently in user interfaces. Instead of saying that a `PictureBox` class is both an `Image` and a `Button`, you might say that it’s an `Image` that is `Clickable`. A folder icon on your desktop could be an `Image` that is `Draggable`. Software developers tend to make up lots of fun adjectives.

The difference between a mixin class and a base class has more to do with how you think about the class than any code difference. In general, mixin classes are easier to digest than multiple inheritance because they are very limited in scope. The `Pettable` mixin class just adds one behavior to any existing class. The `Clickable` mixin class might just add “mouse down” and “mouse up” behaviors. Also, mixin classes rarely have a large hierarchy so there’s no cross-contamination of functionality.

ABSTRACTION

In Chapter 4, you learned about the concept of abstraction — the notion of separating implementation from the means used to access it. Abstraction is a good idea for many reasons explored earlier. It’s also a fundamental part of object-oriented design.

Interface versus Implementation

The key to abstraction is effectively separating the *interface* from the *implementation*. Implementation is the code you're writing to accomplish the task you set out to accomplish. Interface is the way that other people use your code. In C, the header file that describes the functions in a library you've written is an interface. In object-oriented programming, the interface to a class is the collection of publicly accessible properties and behaviors. A good interface contains only public behaviors. Properties/variables of a class should never be made public but can be exposed through public behaviors also called *getters* and *setters*.

Deciding on an Exposed Interface

The question of how other programmers will interact with your objects comes into play when designing a class. In C++, a class's properties and behaviors can each be `public`, `protected`, or `private`. Making a property or behavior `public` means that other code can access it. `protected` means that other code cannot access the property or behavior but derived classes can access them. `private` is a stricter control, which means that not only are the properties or behaviors locked for other code, but even derived classes can't access them.

Designing the exposed interface is all about choosing what to make `public`. When working on a large project with other programmers, you should view the exposed interface design as a process.

Consider the Audience

The first step in designing an exposed interface is to consider for whom you are designing it. Is your audience another member of your team? Is this an interface that you will personally be using? Is it something that a programmer external to your company will use? Perhaps a customer or an off-shore contractor? In addition to determining who will be coming to you for help with the interface, this should shed some light on some of your design goals.

If the interface is for your own use, you probably have more freedom to iterate on the design. As you're making use of the interface, you can change it to suit your own needs. However, you should keep in mind that roles on an engineering team change and it is quite likely that, some day, others will be using this interface as well.

Designing an interface for other internal programmers to use is slightly different. In a way, your interface becomes a contract with them. For example, if you are implementing the data store component of a program, others are depending on that interface to support certain operations. You will need to find out all of the things that the rest of the team wants your class to do. Do they need versioning? What types of data can they store? As a contract, you should view the interface as slightly less flexible. If the interface is agreed upon before coding begins, you'll receive some groans from other programmers if you decide to change it after code has been written.

If the client is an external customer, you will be designing with a very different set of requirements. Ideally, the target customer will be involved in specifying what functionality your interface exposes. You'll need to consider both the specific features they want as well as what customers might want in the future. The terminology used in the interface will have to correspond to the terms that the customer is familiar with, and the documentation will have to be written with that audience in mind. Inside jokes, codenames, and programmer slang should probably be left out of your design.

Consider the Purpose

There are many reasons for writing an interface. Before putting any code on paper or even deciding on what functionality you're going to expose, you need to understand the purpose of the interface.

Application Programming Interface (API)

An *API* is an externally visible mechanism to extend a product or use its functionality within another context. If an internal interface is a contract, an API is closer to a set-in-stone law. Once people who don't even work for your company are using your API, they don't want it to change unless you're adding new features that will help them. So, care should be given to planning the API and discussing it with customers before making it available to them.

The main tradeoff in designing an API is usually ease of use versus flexibility. Because the target audience for the interface is not familiar with the internal working of your product, the learning curve to use the API should be gradual. After all, your company is exposing this API to customers because the company wants it to be used. If it's too difficult to use, the API is a failure. Flexibility often works against this. Your product may have a lot of different uses, and you want the customer to be able to leverage all the functionality you have to offer. However, an API that lets the customer do anything that your product can do may be too complicated.

As a common programming adage goes, “A good API makes the easy case easy and the hard case *possible*.” That is, APIs should have a simple learning curve. The things that most programmers will want to do should be accessible. However, the API should allow for more advanced usage, and it's acceptable to trade off complexity of the rare case for simplicity of the common case.

Utility Class or Library

Often, your task is to develop some particular functionality for general use elsewhere in the application. It could be a random number library or a logging class. In these cases, the interface is somewhat easier to decide on because you tend to expose most or all of the functionality, ideally without giving too much away about its implementation. Generality is an important issue to consider. Since the class or library is general purpose, you'll need to take the possible set of use cases into account in your design.

Subsystem Interface

You may be designing the interface between two major subsystems of the application, such as the mechanism for accessing a database. In these cases, separating the interface from the implementation is paramount because other programmers are likely to start implementing against your interface before your implementation is complete. When working on a subsystem, first think about what its one main purpose is. Once you have identified the main task your subsystem is charged with, think about specific uses and how it should be presented to other parts of the code. Try to put yourself in their shoes and not get bogged down in implementation details.

Component Interface

Most of the interfaces you define will probably be smaller than a subsystem interface or an API. These will be classes that you use within other code that you've written. In these cases, the main pitfall is when your interface evolves gradually and becomes unruly. Even though these interfaces

are for your own use, think of them as though they weren't. As with a subsystem interface, consider the one main purpose of each class and be cautious of exposing functionality that doesn't contribute to that purpose.

Consider the Future

As you are designing your interface, keep in mind what the future holds. Is this a design you will be locked into for years? If so, you might need to leave room for expansion by coming up with a plug-in architecture. Do you have evidence that people will try to use your interface for purposes other than what it was designed for? Talk to them and get a better understanding of their use case. The alternative is rewriting it later, or worse, attaching new functionality haphazardly and ending up with a messy interface. Be careful though! Speculative generality is yet another pitfall. Don't design the be-all end-all logging class if the future uses are unclear, because it might unnecessarily complicate the design, the implementation and its public interface.

Designing a Successful Abstraction

Experience and iteration are essential to good abstractions. Truly well-designed interfaces come from years of writing and using other abstractions. You can also leverage someone else's years of writing and using abstractions by reusing existing, truly well designed abstractions in the form of standard design patterns. As you encounter other abstractions, try to remember what worked and didn't work. What did you find lacking in the Windows file system API you used last week? What would you have done differently if you had written the network wrapper, instead of your coworker? The best interface is rarely the first one you put on paper, so keep iterating. Bring your design to your peers and ask for feedback. If your company uses code reviews, start the code review by doing a review of the interface specifications before the implementation starts. Don't be afraid to change the abstraction once coding has begun, even if it means forcing other programmers to adapt. Hopefully, they'll realize that a good abstraction is beneficial to everyone in the long term.

Sometimes you need to evangelize a bit when communicating your design to other programmers. Perhaps the rest of the team didn't see a problem with the previous design or feels that your approach requires too much work on their part. In those situations, be prepared both to defend your work and to incorporate their ideas when appropriate.

A good abstraction means that the interface has only public behaviors. All code should be in the implementation file and not in the class definition file. This means that the interface files containing the class definitions are stable and will not change.

Beware of single-class abstractions. If there is significant depth to the code you're writing, consider what other companion classes might accompany the main interface. For example, if you're exposing an interface to do some data processing, consider also writing a result object that provides an easy way to view and interpret the results.

Always turn properties into behaviors. In other words, don't allow external code to manipulate the data behind your class directly. You don't want some careless or nefarious programmer to set the height of a bunny object to a negative number. Instead, have a "set height" behavior that does the necessary bounds checking.

Iteration is worth mentioning again because it is the most important point. Seek and respond to feedback on your design, change it when necessary, and learn from mistakes.

SUMMARY

In this chapter, you've gained an appreciation for the design of object-oriented programs without a lot of code getting in the way. The concepts you've learned are applicable in almost any object-oriented language. Some of it may have been a review to you, or it may be a new way of formalizing a familiar concept. Perhaps you picked up some new approaches to old problems or new arguments in favor of the concepts you've been preaching to your team all along. Even if you've never used objects in your code, or have used them only sparingly, you now know more about how to design object-oriented programs than many experienced C++ programmers.

The relationships between objects are important to study, not just because well-linked objects contribute to code reuse and reduce clutter, but also because you will be working in a team. Objects that relate in meaningful ways are easier to read and maintain. You may decide to use the “Object Relationships” section as a reference when you design your programs.

Finally, you learned about creating successful abstractions and the two most important design considerations — audience and purpose.

The next chapter continues the design theme by explaining how to design your code with reuse in mind.

6

Designing for Reuse

WHAT'S IN THIS CHAPTER?

- The reuse philosophy: Why you should design code for reuse
- How to design reusable code
 - How to use abstraction
 - Three strategies for structuring your code for reuse
 - Six strategies for designing usable interfaces
- How to reconcile generality with ease of use

Reusing libraries and other code in your programs is an important design strategy. However, it is only half of the *reuse* strategy. The other half is designing and writing the code that you can reuse in your programs. As you've probably discovered, there is a significant difference between well-designed and poorly designed libraries. Well-designed libraries are a pleasure to use, while poorly designed libraries can prod you to give up in disgust and write the code yourself. Whether you're writing a library explicitly designed for use by other programmers or merely deciding on a class hierarchy, you should design your code with reuse in mind. You never know when you'll need a similar piece of functionality in a subsequent project.

Chapter 4 introduced the design theme of reuse and explained how to apply this theme by incorporating libraries and other code in your designs. This chapter discusses the other side of reuse: designing reusable code. It builds on the object-oriented design principles described in Chapter 5 and introduces some new strategies and guidelines.

THE REUSE PHILOSOPHY

You should design code that both you and other programmers can reuse. This rule applies not only to libraries and frameworks that you specifically intend for other programmers to use, but also to any class, subsystem, or component that you design for a program.

You should always keep in mind the mottos, “write once, use often,” and “avoid code duplication at all cost.” There are several reasons:

- **Code is rarely used in only one program.** You can be sure that your code will be used again somehow, so design it correctly to begin with.
- **Designing for reuse saves time and money.** If you design your code in a way that precludes future use, you ensure that you or your partners will spend time reinventing the wheel later when you encounter a need for a similar piece of functionality.
- **Other programmers in your group must be able to use the code that you write.** You are probably not working alone on a project. Your coworkers will appreciate your efforts to offer them well-designed, functionality-packed libraries and pieces of code to use. Designing for reuse can also be called *cooperative coding*.
- **Lack of reuse leads to code duplication, code duplication leads to a maintenance nightmare.** If ever you find yourself copy-pasting a piece of code, you have to at least consider moving it out to a helper function or class.
- **You will be the primary beneficiary of your own work.** Experienced programmers never throw away code. Over time, they build a personal library of evolving tools. You never know when you will need a similar piece of functionality in the future.

WARNING *When you design or write code as an employee of a company, the company, not you, generally owns the intellectual property rights. It is often illegal to retain copies of your designs or code when you terminate your employment with the company. The same is also true when you are self-employed working for clients.*

HOW TO DESIGN REUSABLE CODE

Reusable code fulfills two main goals. First, it is general enough to use for slightly different purposes or in different application domains. Program components with details of a specific application are difficult to reuse in other programs.

Reusable code is also easy to use. It doesn’t require significant time to understand its interface or functionality. Programmers must be able to incorporate it readily into their applications.

The means of “delivering” your library to clients is also important. You could deliver it in source form and clients just incorporate your source into their project. Another option is to deliver a static library, which they link into their application, or you could deliver a Dynamic Link Library (DLL) for Windows clients or a shared object (.so) for Linux clients. Each of these delivery mechanisms can impose additional constraints on how you code your library.

NOTE This chapter uses the term “client” to refer to a programmer who uses your interfaces. Don’t confuse clients with “users” who run your programs. The chapter also uses the phrase “client code” to refer to code that is written to use your interfaces.

The most important strategy for designing reusable code is abstraction. Chapter 4 presented the real-world analogy of a television, which you can use through its interfaces without understanding how it works inside. Similarly, when you design code, you should clearly separate the interface from the implementation. This separation makes the code easier to use, primarily because clients do not need to understand the internal implementation details in order to use the functionality.

Abstraction separates code into interface and implementation, so designing reusable code focuses on these two main areas. First, you must structure the code appropriately. What class hierarchies will you use? Should you use templates? How should you divide the code into subsystems?

Second, you must design the *interfaces*, which are the “entries” into your library or code that programmers use to access the functionality you provide.

Use Abstraction

You learned about the principle of abstraction in Chapter 4 and read more about its application to object-oriented design in Chapter 5. To follow the principle of abstraction, you should provide interfaces to your code that hide the underlying implementation details. There should be a clear distinction between the interface and the implementation.

Using abstraction benefits both you and the clients who use your code. Clients benefit because they don’t need to worry about the implementation details; they can take advantage of the functionality you offer without understanding how the code really works. You benefit because you can modify the underlying code without changing the interface to the code. Thus, you can provide upgrades and fixes without requiring clients to change their use. With dynamically linked libraries, clients might not even need to rebuild their executables. Finally, you both benefit because you, as the library writer, can specify in the interface exactly what interactions you expect and functionality you support. Consult Chapter 3 for a discussion on how to write documentation. A clear separation of interfaces and implementations will prevent clients from using the library in ways that you didn’t intend, which can otherwise cause unexpected behaviors and bugs.

WARNING When designing your interface, do not expose implementation details to your clients.

Sometimes libraries require client code to keep information returned from one interface in order to pass it to another. This information is sometimes called a *handle* and is often used to keep track of specific instances that require state to be remembered between calls. If your library design requires a handle, don’t expose its internals. Make that handle into an *opaque* class, in which the programmer “can’t access the internal data members, neither directly, nor through public getters or setters.”

Don't require the client code to tweak variables inside this handle. An example of a bad design would be a library that requires you to set a specific member of a structure in a supposedly opaque handle in order to turn on error logging.

NOTE *Unfortunately, C++ is fundamentally unfriendly to the principle of good abstraction when writing classes. The syntax requires you to combine your public interfaces and non-public (private or protected) data members and methods together in one class definition, thereby exposing some of the internal implementation details of the class to its clients. Chapter 8 describes some techniques for working around this in order to present clean interfaces.*

Abstraction is so important that it should guide your entire design. As part of every decision you make, ask yourself whether your choice fulfills the principle of abstraction. Put yourself in your clients' shoes and determine whether or not you're requiring knowledge of the internal implementation in the interface. You should rarely, if ever, make exceptions to this rule.

Structure Your Code for Optimal Reuse

You must consider reuse from the beginning of your design. The following strategies will help you organize your code properly. Note that all of these strategies focus on making your code general purpose. The second aspect of designing reusable code, providing ease of use, is more relevant to your interface design and is discussed later in this chapter.

Avoid Combining Unrelated or Logically Separate Concepts

When you design a library or framework, keep it focused on a single task or group of tasks, i.e., strive for *high cohesion*. Don't combine unrelated concepts such as a random number generator and an XML parser.

Even when you are not designing code specifically for reuse, keep this strategy in mind. Entire programs are rarely reused on their own. Instead, pieces or subsystems of the programs are incorporated directly into other applications, or are adapted for slightly different uses. Thus, you should design your programs so that you divide logically separate functionality into distinct components that can be reused in different programs.

This program strategy models the real-world design principle of discrete, interchangeable parts. For example, you could take the tires off an old car and use them on a new car of a different model. Tires are separable components that are not tied to other aspects of the car. You don't need to bring the engine along with the tires!

You can employ the strategy of logical division in your program design on both the macro subsystem level and the micro class hierarchy level.

Divide Your Programs into Logical Subsystems

Design your subsystems as discrete components that can be reused independently, i.e., strive for *low coupling*. For example, if you are designing a networked game, keep the networking and graphical

user interface aspects in separate subsystems. That way you can reuse either component without dragging in the other. For example, you might want to write a non-networked game, in which case you could reuse the graphical interface subsystem, but wouldn't need the networking aspect. Similarly, you could design a peer-to-peer file-sharing program, in which case you could reuse the networking subsystem but not the graphical user interface functionality.

Make sure to follow the principle of abstraction for each subsystem. Think of each subsystem as a miniature library for which you must provide a coherent and easy-to-use interface. Even if you're the only programmer who ever uses these miniature libraries, you will benefit from well-designed interfaces and implementations that separate logically distinct functionality.

Use Class Hierarchies to Separate Logical Concepts

In addition to dividing your program into logical subsystems, you should avoid combining unrelated concepts at the class level. For example, suppose you want to write a balanced binary tree structure for a multithreaded program. You decide that the tree data structure should allow only one thread at a time to access or modify the structure, so you incorporate locking into the data structure itself. However, what if you want to use this binary tree in another program that happens to be single-threaded? In that case, the locking is a waste of time, and might require your program to link with libraries that it could otherwise avoid. Even worse, your tree structure might not compile on a different platform because the locking code might not be cross-platform. A solution is to create a class hierarchy (introduced in Chapter 5) in which a thread-safe binary tree is a derived class of a generic binary tree. That way you can use the binary tree base class in single-threaded programs without incurring the cost of locking unnecessarily. Figure 6-1 shows this hierarchy.

This strategy works well when there are two logical concepts, such as thread safety and binary trees. It becomes more complicated when there are three or more concepts. For example, suppose you want to provide both an n -ary tree and a binary tree, each of which could be thread-safe or not. Logically, the binary tree is a special-case of an n -ary tree, and so should be a derived class of an n -ary tree. Similarly, thread-safe structures could be derived classes of non-thread-safe structures. You can't provide these separations with a linear hierarchy. One possibility is to make the thread-safe aspect a mix-in class as shown in Figure 6-2.

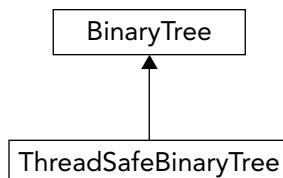


FIGURE 6-1

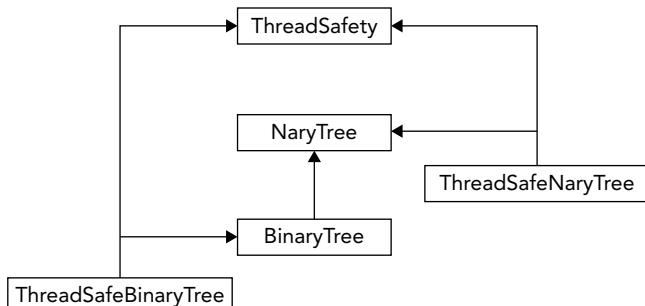


FIGURE 6-2

That hierarchy requires you to write five different classes, but the clear separation of functionality is worth the effort.

You should also avoid combining unrelated concepts at the level of methods, just as you should do at the class level. Both at the class level and the method level you should strive for high cohesion. For example, methods should not mix mutation (set) and inspection (get).

Use Aggregation to Separate Logical Concepts

Aggregation, discussed in Chapter 5, models the *has-a* relationship: Objects contain other objects to perform some aspects of their functionality. You can use aggregation to separate unrelated or related but separate functionality when inheritance is not appropriate.

For example, suppose you want to write a `Family` class to store the members of a family. Obviously, a tree data structure would be ideal for storing this information. Instead of integrating the code for the tree structure in your `Family` class, you should write a separate `Tree` class. Your `Family` class can then contain and use a `Tree` instance. To use the object-oriented terminology, the `Family` has-a `Tree`. With this technique, the tree data structure could be reused more easily in another program.

Eliminate User Interface Dependencies

If your library is a data manipulation library, you want to separate data manipulation from the user interface. This means that for those kinds of libraries you should never assume in which type of user interface the library will be used. As such, do not use `cout`, `cerr`, `cin`, `stdout`, `stderr`, or `stdin`, because if the library is used in the context of a graphical user interface these concepts may make no sense. For example, a Windows GUI-based application normally will not have any form of console I/O. If you think your library will only be used in GUI-based applications, you should still never pop up any kind of message box or other kind of notification to the end user; that is the responsibility of the client code to do. These kinds of dependencies not only result in poor reusability, but they prevent the client code from properly responding to an error, and handling it silently.

The Model-View-Controller (MVC) paradigm, introduced in Chapter 4, is a well-known design pattern to separate storing data from visualizing the data. With this paradigm, the model can be in the library, while the client code can provide the view and the controller.

Use Templates for Generic Data Structures and Algorithms

C++ has a concept called *templates* allowing you to create structures that are generic with respect to a type or class. For example, you might have written code for an array of integers. If you subsequently would like an array of doubles, you need to rewrite and replicate all the code to work with doubles. The notion of a template is that the type becomes a parameter to the specification, and you can create a single body of code that can work on any type. Templates allow you to write both data structures and algorithms that work on any types.

The simplest example of this is the `std::vector` class which is part of the C++ Standard Template Library (STL). To create a vector of integers, you write `std::vector<int>`; to create a vector of doubles you write `std::vector<double>`. Template programming is in general extremely powerful but can be very complex. Luckily, it is possible to create rather simple usages of templates that parameterize according to a type. Chapters 11 and 21 explain the techniques to write your own templates, while this section discusses some of their important design aspects.

Whenever possible, you should use a generic design for data structures and algorithms instead of encoding specifics of a particular program. Don't write a balanced binary tree structure that stores

only book objects. Make it generic, so that it can store objects of any type. That way you could use it in a bookstore, a music store, an operating system, or anywhere that you need a balanced binary tree. This strategy underlies the standard template library. The STL provides generic data structures and algorithms that work on any types.

Why Templates Are Better Than Other Generic Programming Techniques

Templates are not the only mechanism for writing generic data structures. You can write generic structures in C and C++ by storing `void*` pointers instead of a specific type. Clients can use this structure to store anything they want by casting it to a `void*`. However, the main problem with this approach is that it is not *type-safe*: the containers are unable to check or enforce the types of the stored elements. You can cast any type to a `void*` to store in the structure, and when you remove the pointers from the data structure, you must cast them back to what you think they are. Because there are no checks involved, the results can be disastrous. Imagine a scenario where one programmer stores pointers to `int` in a data structure by first casting them to `void*`, but another programmer thinks they are pointers to `Process` objects. The second programmer will blithely cast the `void*` pointers to `Process*` pointers and try to use them as `Process*`s. Needless to say, the program will not work as expected.

A second approach is to write the data structure for a specific class. Through polymorphism, any derived class of that class can be stored in the structure. Java takes this approach to an extreme: It specifies that every class derives directly or indirectly from the `Object` class. The Java containers store objects, so they can store objects of any type. However, this approach is also not truly type-safe. When you remove an object from the container, you must remember what it really is and down-cast it to the appropriate type.

Templates, on the other hand, are type-safe when used correctly. Each instantiation of a template stores only one type. Your program will not compile if you try to store different types in the same template instantiation. Java does support the concept of generics which are type-safe just as C++ templates. Java does support the concept of *generics* which are type-safe just as C++ templates.

Problems with Templates

Templates are not perfect. First of all, their syntax is confusing, especially for someone who has not used them before. Second, the parsing is difficult, and not all compilers fully support the entire C++ standard. If your compiler does not fully support the C++ standard regarding templates, your compiler might also not support the entire feature set of the STL. On the other hand, any compiler that fully supports the STL should be sufficiently powerful to support most template programming.

Furthermore, templates require homogeneous data structures, in which you can store only objects of the same type in a single structure. That is, if you write a templated balanced binary tree, you can create one tree object to store `Process` objects and another tree object to store `int`s. You can't store both `int`s and `Processes` in the same tree. This restriction is a direct result of the type-safe nature of templates.

Templates versus Inheritance

Programmers sometimes find it tricky to decide whether to use templates or inheritance. Here are some tips to help you make the decision.

Use templates when you want to provide identical functionality for different types. For example, if you want to write a generic sorting algorithm that works on any type, use templates. If you want

to create a container that can store any type, use templates. The key concept is that the templated structure or algorithm treats all types the same. However, if required, templates can be specialized for specific types to treat those types differently. Template specialization is discussed in Chapter 11.

When you want to provide different behaviors for related types, use inheritance. For example, use inheritance if you want to provide two different, but similar, containers such as a queue and a priority queue.

Note that you can combine inheritance and templates. You could write a templated queue that stores any type, with a derived class that is a templated priority queue. Chapter 11 covers the details of the template syntax.

Provide Appropriate Checks and Safeguards

There are two opposite styles for writing safe code. The optimal programming style is probably using a healthy mix of both of them. The first is called *design-by-contract* which means that the documentation for a function or a class represents a contract with a detailed description of what the responsibility of the client code is and what the responsibility of your function or class is. This is often used in the STL. For example, `std::vector` defines a contract for using the array notation to get a certain element from a vector. The contract says that the `vector` will not perform any bounds checking, but that this is the responsibility of the client code. This is done to increase performance for client code that knows their indices are within bounds. The `vector` also defines an `at()` method to get a specific element which does perform bounds checking. So client code can choose whether it uses the array notation without, or the `at()` method with bounds checking.

The second style is that you design your functions and classes to be as safe as possible. The most important aspect of this guideline is to perform error checking in your code. For example, if your random number generator requires a non-negative integer for a seed, don't just trust the user to correctly pass a non-negative integer. Check the value that is passed in, and reject the call if it is invalid.

As an analogy, consider an accountant who prepares income tax returns. When you hire an accountant, you provide him or her with all your financial information for the year. The accountant uses this information to fill out forms from the IRS. However, the accountant does not blindly fill out your information on the form, but instead makes sure the information makes sense. For example, if you own a house, but forget to specify the property tax you paid, the accountant will remind you to supply that information. Similarly, if you say that you paid \$12,000 in mortgage interest, but made only \$15,000 gross income, the accountant might gently ask you if you provided the correct numbers (or at least recommend more affordable housing).

You can think of the accountant as a “program” where the input is your financial information and the output is an income tax return. However, the value added by an accountant is not just that he or she fills out the forms. You choose to employ an accountant also because of the checks and safeguards that he or she provides. Similarly in programming, you could provide as many checks and safeguards as possible in your implementations.

There are several techniques and language features that help you incorporate checks and safeguards in your programs. First, you can return an error code or a distinct value like `false` or `nullptr` or throw an exception to notify the client code of an error. Chapter 13 covers exceptions in detail. Second, use smart pointers that help you manage your dynamically allocated memory. Conceptually, a smart pointer is a pointer to dynamically allocated memory that remembers to free the memory when it goes out of scope. Third, use safe memory techniques as discussed in Chapter 22.

Design Usable Interfaces

In addition to abstracting and structuring your code appropriately, designing for reuse requires you to focus on the *interface* with which programmers interact. If you have the most beautiful and most efficient implementation, your library will not be any good if it has a wretched interface.

Note that every subsystem and class in your program should have good interfaces, even if you don't intend them to be used in multiple programs. First of all, you never know when something will be reused. Second, a good interface is important even for the first use, especially if you are programming in a group and other programmers must use the code you design and write.

The main purpose of interfaces is to make the code easy to use, but some interface techniques can help you follow the principle of generality as well.

Design Interfaces That Are Easy To Use

Your interfaces should be easy to use. That doesn't mean that they must be trivial, but they should be as simple and intuitive as the functionality allows. You shouldn't require consumers of your library to wade through pages of source code in order to use a simple data structure, or go through contortions in their code to obtain the functionality they need. This section provides four specific strategies for designing interfaces that are easy to use.

Develop Easy-To-Use Interfaces

The best strategy for developing easy-to-use interfaces is to follow standard and familiar ways of doing things. When people encounter an interface similar to something they have used in the past, they will understand it better, adopt it more readily, and be less likely to use it improperly.

For example, suppose that you are designing the steering mechanism of a car. There are a number of possibilities: a joystick, two buttons for moving left or right, a sliding horizontal lever, or a good-old steering wheel. Which interface do you think would be easiest to use? Which interface do you think would sell the most cars? Consumers are familiar with steering wheels, so the answer to both questions is, of course, the steering wheel. Even if you developed another mechanism that provided superior performance and safety, you would have a tough time selling your product, let alone teaching people how to use it. When you have a choice between following standard interface models and branching out in a new direction, it's usually better to stick to the interface to which people are accustomed.

Innovation is important, of course, but you should focus on innovation in the underlying implementation, not in the interface. For example, consumers are excited about the innovative hybrid gasoline-electric engine in some car models. These cars are selling well in part because the interface to use them is identical to cars with standard engines.

Applied to C++, this strategy implies that you should develop interfaces that follow standards to which C++ programmers are accustomed. For example, C++ programmers expect the constructor and destructor of a class to initialize and clean up an object, respectively. When you design your classes, you should follow this standard. If you require programmers to call `initialize()` and `cleanup()` methods for initialization and cleanup instead of placing that functionality in the constructor and destructor, you will confuse everyone who tries to use your class. Because your class behaves differently from other C++ classes, programmers will take longer to learn how to use it and will be more likely to use it incorrectly by forgetting to call `initialize()` or `cleanup()`.

NOTE Always think about your interfaces from the perspective of someone using them. Do they make sense? Are they what you would expect?

C++ provides a language feature called *operator overloading* that can help you develop easy-to-use interfaces for your objects. Operator overloading allows you to write classes such that the standard operators work on them just as they work on built-in types like `int` and `double`. For example, you can write a `Fraction` class that allows you to add, subtract, and stream fractions like this:

```
Fraction f1(3,4);
Fraction f2(1,2);
Fraction sum;
Fraction diff;
sum = f1 + f2;
diff = f1 - f2;
cout << f1 << " " << f2 << endl;
```

Contrast that with the same behavior using method calls:

```
Fraction f1(3,4);
Fraction f2(1,2);
Fraction sum;
Fraction diff;
sum = f1.add(f2);
diff = f1.subtract(f2);
f1.print(cout);
cout << " ";
f2.print(cout);
cout << endl;
```

As you can see, operator overloading allows you to provide an easier to use interface for your classes. However, be careful not to abuse operator overloading. It's possible to overload the `+` operator so that it implements subtraction and the `-` operator so that it implements multiplication. Those implementations would be counterintuitive. This does not mean that each operator should always implement exactly the same behavior. For example, the `string` class implements the `+` operator to concatenate `strings`, which is an intuitive interface for `string` concatenation. See Chapters 8 and 14 for details on operator overloading.

Don't Omit Required Functionality

This strategy is twofold. First, include interfaces for all behaviors that clients could need. That might sound obvious at first. Returning to the car analogy, you would never build a car without a speedometer for the driver to view his or her speed! Similarly, you would never design a `Fraction` class without a mechanism for client code to access the nominator and denominator values.

However, other possible behaviors might be more obscure. This strategy requires you to anticipate all the uses to which clients might put your code. If you are thinking about the interface in one

particular way, you might miss functionality that could be needed when clients use it differently. For example, suppose that you want to design a game board class. You might consider only the typical games, such as chess and checkers, and decide to support a maximum of one game piece per spot on the board. However, what if you later decide to write a backgammon game, which allows multiple pieces in one spot on the board? By precluding that possibility, you have ruled out the use of your game board as a backgammon board.

Obviously, anticipating every possible use for your library is difficult, if not impossible. Don't feel compelled to agonize over potential future uses in order to design the perfect interface. Just give it some thought and do the best you can.

The second part of this strategy is to include as much functionality in the implementation as possible. Don't require client code to specify information that you already know in the implementation, or could know if you designed it differently. For example, if your library requires a temporary file, don't make the clients of your library specify that path. They don't care what file you use; find some other way to determine an appropriate temporary file path.

Furthermore, don't require library users to perform unnecessary work to amalgamate results. If your random number library uses a random number algorithm that calculates the low-order and high-order bits of a random number separately, combine the numbers before giving them to the user.

Present Uncluttered Interfaces

In order to avoid omitting functionality in their interfaces, some programmers go to the opposite extreme: They include every possible piece of functionality imaginable. Programmers who use the interfaces are never left without the means to accomplish a task. Unfortunately, the interface might be so cluttered that they never figure out how to do it!

Don't provide unnecessary functionality in your interfaces; keep them clean and simple. It might appear at first that this guideline directly contradicts the previous strategy of avoiding omitting necessary functionality. Although one strategy to avoid omitting functionality would be to include every imaginable interface, that is not a sound strategy. You should include *necessary* functionality and omit useless or counterproductive interfaces.

Consider cars again. You drive a car by interacting with only a few components: the steering wheel, the brake and accelerator pedals, the gearshift, the mirrors, the speedometer, and a few other dials on your dashboard. Now, imagine a car dashboard that looked like an airplane cockpit, with hundreds of dials, levers, monitors, and buttons. It would be unusable! Driving a car is so much easier than flying an airplane that the interface can be much simpler: You don't need to view your altitude, communicate with control towers, or control the myriad components in an airplane such as the wings, engines, and landing gear.

Additionally, from the library development perspective, smaller libraries are easier to maintain. If you try to make everyone happy, then you have more room to make mistakes, and if your implementation is complicated enough so that everything is intertwined, even one mistake can render the library useless.

Unfortunately, the idea of designing uncluttered interfaces looks good on paper, but is remarkably hard to put into practice. The rule is ultimately subjective: You decide what's necessary and what's not. Of course, your clients will be sure to tell you when you get it wrong!

Provide Documentation and Comments

Regardless of how easy to use you make your interfaces, you should supply documentation for their use. You can't expect programmers to use your library properly unless you tell them how to do it. Think of your library or code as a product for other programmers to consume. Your product should have documentation explaining its proper use.

There are two ways to provide documentation for your interfaces: comments in the interfaces themselves and external documentation. You should strive to provide both. Most public APIs provide only external documentation: Comments are a scarce commodity in many of the standard Unix and Windows header files. In Unix, the documentation usually comes in the form of online manuals called *man pages*. In Windows, the documentation usually accompanies the integrated development environment.

Despite the fact that most APIs and libraries omit comments in the interfaces themselves, I actually consider this form of documentation the most important. You should never give out a “naked” header file that contains only code. Even if your comments repeat exactly what's in the external documentation, it is less intimidating to look at a header file with friendly comments than one with only code. Even the best programmers still like to see written language every so often!

Some programmers use tools to create documentation automatically from comments. Chapter 3 discusses this technique in more detail.

Whether you provide comments, external documentation, or both, the documentation should describe the *behavior* of the library, not the *implementation*. The behavior includes the inputs, outputs, error conditions and handling, intended uses, and performance guarantees. For example, documentation describing a call to generate a single random number should specify that it takes no parameters, returns an integer in a previously specified range, and should list all the exceptions that might be thrown when something goes wrong. This documentation should not explain the details of the linear congruence algorithm for actually generating the number. Providing too much implementation detail in interface comments is probably the single most common mistake in interface development. Many developers have seen perfectly good separations of interface and implementation ruined by comments in the interface that are more appropriate for library maintainers than clients.

Of course you should also document your internal implementation, just don't make it publicly available as part of your interface. Chapter 3 provides details on the appropriate use of comments in your code.

Design General-Purpose Interfaces

The interfaces should be general purpose enough that they can be adapted to a variety of tasks. If you encode specifics of one application in a supposedly general interface, it will be unusable for any other purpose. Here are some guidelines to keep in mind.

Provide Multiple Ways to Perform the Same Functionality

In order to satisfy all your “customers,” it is sometimes helpful to provide multiple ways to perform the same functionality. Use this technique judiciously, however, because overapplication can easily lead to cluttered interfaces.

Consider cars again. Most new cars these days provide remote keyless entry systems, with which you can unlock your car by pressing a button on a key fob. However, these cars always provide a standard key that you can use to physically unlock the car, for example when the battery in the key fob is empty. Although these two methods are redundant, most customers appreciate having both options.

Sometimes there are similar situations in program interface design. For example, suppose that one of your methods takes a string. You might want to provide two interfaces: one that takes a C++ `string` object and one that takes a C-style character pointer. Although it's possible to convert between the two, different programmers prefer different types of strings, so it's helpful to cater to both approaches.

Note that this strategy should be considered an exception to the “uncluttered” rule in interface design. There are a few situations where the exception is appropriate, but you should most often follow the “uncluttered” rule.

Provide Customizability

In order to increase the flexibility of your interfaces, provide customizability. Customizability can be as simple as allowing a client to turn on or off error logging. The basic premise of customizability is that it allows you to provide the same basic functionality to every client, but gives clients the ability to tweak it slightly.

You can allow greater customizability through function pointers and template parameters. For example, you could allow clients to set their own error-handling routines. This technique is an application of the decorator pattern.

The STL takes this customizability strategy to the extreme and actually allows clients to specify their own memory allocators for containers. If you want to use this feature, you must write a memory allocator object that follows the STL guidelines and adheres to the required interfaces. Each container in the STL takes an allocator as one of its template parameters. Chapter 20 provides more details.

Reconciling Generality and Ease of Use

The two goals of ease of use and generality sometimes appear to conflict. Often, introducing generality increases the complexity of the interfaces. For example, suppose that you need a graph structure in a map program to store cities. In the interest of generality, you might use templates to write a generic map structure for any type, not just cities. That way, if you need to write a network simulator in your next program, you could employ the same graph structure to store routers in the network. Unfortunately, by using templates, you made the interface a little clumsier and harder to use, especially if the potential client is not familiar with templates.

However, generality and ease of use are not mutually exclusive. Although in some cases increased generality may decrease ease of use, it is possible to design interfaces that are both general purpose and straightforward to use. Here are two guidelines you can follow.

Supply Multiple Interfaces

In order to reduce complexity in your interfaces while still providing enough functionality, you can provide two separate interfaces. For example, you could write a generic networking library with

two separate facets: One presents the networking interfaces useful for games, and one presents the networking interfaces useful for the HyperText Transport Protocol (HTTP) for web browsing.

Make Common Functionality Easy To Use

When you provide a general-purpose interface, some functionality will be used more often than other functionality. You should make the commonly used functionality easy to use, while still providing the option for the more advanced functionality. Returning to the map program, you might want to provide an option for clients of the map to specify names of cities in different languages. English is so predominant that you could make that the default but provide an extra option to change languages. That way most clients will not need to worry about setting the language, but those who want to will be able to do so.

SUMMARY

By reading this chapter, you learned *why* you should design reusable code and *how* you should do it. You read about the philosophy of reuse, summarized as “write once, use often,” and learned that reusable code should be both general purpose and easy to use. You also discovered that designing reusable code requires you to use abstraction, to structure your code appropriately, and to design good interfaces.

This chapter presented three specific tips for structuring your code: Avoid combining unrelated or logically separate concepts, use templates for generic data structures and algorithms, and provide appropriate checks and safeguards.

The chapter also presented six strategies for designing interfaces: Develop easy-to-use interfaces, don’t omit required functionality, present uncluttered interfaces, provide documentation and comments, provide multiple ways to perform the same functionality, and provide customizability. It concluded with two tips for reconciling the often-conflicting demands of generality and ease of use: Supply multiple interfaces and make common functionality easy to use.

This chapter concludes the second part of this book which focuses on discussing design themes on a higher level. The next part delves into the implementation phase of the software engineering process with details of C++ coding.

PART III

Coding the Professional Way

- **CHAPTER 7:** Gaining Proficiency with Classes and Objects
- **CHAPTER 8:** Mastering Classes and Objects
- **CHAPTER 9:** Discovering Inheritance Techniques
- **CHAPTER 10:** C++ Quirks, Oddities, and Incidentals
- **CHAPTER 11:** Writing Generic Code with Templates
- **CHAPTER 12:** Demystifying C++ I/O
- **CHAPTER 13:** Handling Errors
- **CHAPTER 14:** Overloading C++ Operators
- **CHAPTER 15:** Overview of the C++ Standard Library
- **CHAPTER 16:** Understanding Containers and Iterators
- **CHAPTER 17:** Mastering STL Algorithms
- **CHAPTER 18:** String Localization and Regular Expressions
- **CHAPTER 19:** Additional Library Utilities

7

Gaining Proficiency with Classes and Objects

WHAT'S IN THIS CHAPTER?

- How to write your own classes with methods and data members
- How to control access to your methods and data members
- How to use objects on the stack and on the heap
- What the life cycle of an object is
- How to write code that is executed when an object is created or destroyed
- How to write code to copy or assign objects

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

As an object-oriented language, C++ provides facilities for using objects and for writing object definitions, called classes. You can certainly write programs in C++ without classes and objects, but by doing so, you do not take advantage of the most fundamental and useful aspect of the language; writing a C++ program without classes is like traveling to Paris and eating at McDonald's. In order to use classes and objects effectively, you must understand their syntax and capabilities.

Chapter 1 reviewed the basic syntax of class definitions. Chapter 5 introduced the object-oriented approach to programming in C++ and presented specific design strategies for classes

and objects. This chapter describes the fundamental concepts involved in using classes and objects, including writing class definitions, defining methods, using objects on the stack and the heap, writing constructors, default constructors, compiler-generated constructors, constructor initializers (known as *ctor-initializers*), copy constructors, initializer-list constructors, destructors, and assignment operators. Even if you are already comfortable with classes and objects, you should skim this chapter because it contains various tidbits of information with which you might not yet be familiar.

INTRODUCING THE SPREADSHEET EXAMPLE

This chapter and the next present a runnable example of a simple spreadsheet application. A spreadsheet is a two-dimensional grid of “cells,” and each cell contains a number or string. Professional spreadsheets such as Microsoft Excel provide the ability to perform mathematical operations, such as calculating the sum of the values of a set of cells. The spreadsheet example in these chapters does not attempt to challenge Microsoft in the marketplace, but is useful for illustrating the issues of classes and objects.

The spreadsheet application uses two basic classes: `Spreadsheet` and `SpreadsheetCell`. Each `Spreadsheet` object contains `SpreadsheetCell` objects. In addition, a `SpreadsheetApplication` class manages a collection of `Spreadsheets`. This chapter focuses on the `SpreadsheetCell`. Chapter 8 develops the `Spreadsheet` and `SpreadsheetApplication` classes.

NOTE *This chapter shows several different versions of the `SpreadsheetCell` class in order to introduce concepts gradually. Thus, the various attempts at the class throughout the chapter do not always illustrate the “best” way to do every aspect of class writing. In particular, the early examples omit important features that would normally be included, but have not yet been introduced.*
You can download the final version of the class as described in the Introduction.

WRITING CLASSES

When you write a class you specify the behaviors, or *methods*, that will apply to objects of that class and the properties, or *data members*, that each object will contain.

There are two components in the process of writing classes: defining the classes themselves and defining their methods.

Class Definitions

Here is a first attempt at a simple `SpreadsheetCell` class, in which each cell can store only a single number:

```
class SpreadsheetCell
{
    public:
```

```

    void setValue(double inValue);
    double getValue() const;
private:
    double mValue;
};

```

As described in Chapter 1, every class definition begins with the keyword `class` and the name of the class. A class definition is a *statement* in C++, so it must end with a semicolon. If you fail to terminate your class definition with a semicolon, your compiler will probably give you several errors, most of which will appear to be completely unrelated.

Class definitions usually go in a file named after the class. For example, the `SpreadsheetCell` class definition can be put in a file called `SpreadsheetCell.h`. This rule is not enforced and you are free to call your file however you like.

Class Members

A class can have a number of *members*. A member is either a *member function* (which in turn is either a *method*, *constructor*, or *destructor*), or a *member variable* also called *data member*.

The two lines that look like function prototypes declare the methods that this class supports:

```

void setValue(double inValue);
double getValue() const;

```

Chapter 1 points out that it is always a good idea to declare member functions that do not change the object as `const`.

The line that looks like a variable declaration declares the data member for this class.

```
double mValue;
```

A class defines the member functions and data members that apply. They apply only to a specific *instance* of the class, which is an *object*. The only exception to this rule are static members, explained in Chapter 8. Classes define concepts; objects contain real bits. So, each object will contain its own value for the `mValue` variable. The implementation of the member functions is shared across all objects. Classes can contain any number of member functions and data members. You cannot give a data member the same name as a member function.

Access Control

Every member in a class is subject to one of three *access specifiers*: `public`, `protected`, or `private`. An access specifier applies to all member declarations that follow it, until the next access specifier. In the `SpreadsheetCell` class, the `setValue()` and `getValue()` methods have `public` access, while the `mValue` data member has `private` access.

The default access specifier for classes is `private`: all member declarations before the first access specifier have the `private` access specification. For example, moving the `public` access specifier below the `setValue()` method declaration gives the `setValue()` method `private` access instead of `public`:

```
class SpreadsheetCell
{
    void setValue(double inValue); // now has private access
public:
    double getValue() const;
private:
    double mValue;
};
```

In C++, a `struct` can have methods just like a `class`. In fact, the only difference is that the default access specifier for a `struct` is `public` while the default for a `class` is `private`. For example, the `SpreadsheetCell` class can be rewritten using a `struct` as follows:

```
struct SpreadsheetCell
{
    void setValue(double inValue);
    double getValue() const;
private:
    double mValue;
};
```

The following table summarizes the meanings of the three access specifiers:

ACCESS SPECIFICATION	MEANING	WHEN TO USE
<code>public</code>	Any code can call a <code>public</code> member function or access a <code>public</code> data member of an object.	Behaviors (methods) that you want clients to use. Access methods for <code>private</code> and <code>protected</code> data members.
<code>protected</code>	Any member function of the class can call <code>protected</code> member functions and access <code>protected</code> data members. Member functions of a derived class can access <code>protected</code> members of a base class.	“Helper” methods that you do not want clients to use.
<code>private</code>	Only member functions of the class can call <code>private</code> member functions and access <code>private</code> data members. Member functions in derived classes cannot access <code>private</code> members from a base class.	Everything should be <code>private</code> by default, especially data members. You can provide <code>protected</code> getters and setters if you only want to allow derived classes to access them, and provide <code>public</code> getters and setters if you want clients to access them.

Order of Declarations

You can declare your members and access control specifiers in any order: C++ does not impose any restrictions, such as member functions before data members or `public` before `private`. Additionally, you can repeat access specifiers. For example, the `SpreadsheetCell` definition could look like the following:

```
class SpreadsheetCell
{
    public:
        void setValue(double inValue);
    private:
        double mValue;
    public:
        double getValue() const;
};
```

However, for clarity it is a good idea to group `public`, `protected`, and `private` declarations, and to group member functions and data members within those declarations.

Defining Methods

The preceding definition for the `SpreadsheetCell` class is enough for you to create objects of the class. However, if you try to call the `setValue()` or `getValue()` methods, your linker will complain that those methods are not defined. That's because the class definition specifies the prototypes for the methods, but does not define their implementations. Just as you write both a prototype and a definition for a stand-alone function, you must write a prototype and a definition for a method.

Note that the class definition must precede the method definitions. Usually the class definition goes in a header file, and the method definitions go in a source file that includes that header. Here are the definitions for the two methods of the `SpreadsheetCell` class:

```
#include "SpreadsheetCell.h"
void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
}
double SpreadsheetCell::getValue() const
{
    return mValue;
}
```

Note that the name of the class followed by two colons precedes each method name:

```
void SpreadsheetCell::setValue(double inValue)
```

The `::` is called the *scope resolution operator*. In this context, the syntax tells the compiler that the coming definition of the `setValue()` method is part of the `SpreadsheetCell` class. Note also that you do not repeat the access specification when you define the method.

NOTE *If you are using the Microsoft Visual C++ IDE, you will notice that by default all .cpp files start with:*

```
#include "stdafx.h"
```

In a VC++ project, by default, every .cpp file should start with this line and your own includes must follow this. If you place your own includes before the stdafx.h include, they will appear to have no effect and you will get all kinds of compilation errors.

This deals with the concept of precompiled header files, which is outside the scope of this book. Consult the Microsoft documentation on precompiled header files to learn the details.

Accessing Data Members

Non-static methods of a class, such as `setValue()` and `getValue()`, are always executed on behalf of a specific object of that class. Inside the method body, you have access to all the data members of the class for that object. In the previous definition for `setValue()`, the following line changes the `mValue` variable inside whatever object calls the method:

```
mValue = inValue;
```

If `setValue()` is called for two different objects, the same line of code (executed once for each object) changes the variable in two different objects.

Calling Other Methods

You can call methods of a class from inside another method. For example, consider an extension to the `SpreadsheetCell` class. Real spreadsheet applications allow text data as well as numbers in the cells. When you try to interpret a text cell as a number, the spreadsheet tries to convert the text to a number. If the text does not represent a valid number, the cell value is ignored. In this program, strings that are not numbers will generate a cell value of 0. Here is a first stab at a class definition for a `SpreadsheetCell` that supports text data:

```
#include <string>
class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue() const;
    void setString(const std::string& inString);
    const std::string& getString() const;
private:
    std::string doubleToString(double inValue) const;
    double stringtoDouble(const std::string& inString) const;
    double mValue;
    std::string mString;
};
```

This version of the class stores both text and numerical representations of the data. If the client sets the data as a `string`, it is converted to a `double`, and a `double` is converted to a `string`. If the text is not a valid number, the `double` value is 0. Note that having both a text and a numerical representation in this class is only for illustrative purposes. You should avoid storing redundant data. This deficiency of the `SpreadsheetCell` class is resolved in Chapter 9. This class definition shows two new methods to set and retrieve the text representation of the cell and two new private *helper methods* to convert a `double` to a `string` and vice versa. These helper methods use string streams, which are covered in detail in Chapter 12. Here are the implementations of all the methods.

```
#include "SpreadsheetCell.h"
#include <iostream>
#include <sstream>
using namespace std;
void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
    mString = doubleToString(mValue);
}
double SpreadsheetCell::getValue() const
{
    return mValue;
}
void SpreadsheetCell::setString(const string& inString)
{
    mString = inString;
    mValue = stringtoDouble(mString);
}
const string& SpreadsheetCell::getString() const
{
    return mString;
}
string SpreadsheetCell::doubleToString(double inValue) const
{
    ostringstream ostr;
    ostr << inValue;
    return ostr.str();
}
double SpreadsheetCell::stringtoDouble(const string& inString) const
{
    double temp;
    istringstream istr(inString);
    istr >> temp;
    if (istr.fail() || !istr.eof()) {
        return 0;
    }
    return temp;
}
```

Note that each of the set methods calls a helper method to perform a conversion. With this technique, both `mValue` and `mString` are always valid.

The this Pointer

Every normal method call passes a pointer to the object for which it is called as a “hidden” parameter with the name `this`. You can use this pointer to access data members or call methods, and can pass it to other methods or functions. It is also sometimes useful for disambiguating names. For example, you could have defined the `SpreadsheetCell` class with a `value` data member instead of `mValue` and you could have defined the `setValue()` method to take a parameter named `value` instead of `inValue`. In that case, `setValue()` would look like this:

```
void SpreadsheetCell::setValue(double value)
{
    value = value; // Ambiguous!
    mString = doubleToString(value);
}
```

That line is confusing. Which `value` do you mean: the `value` that was passed as a parameter, or the `value` that is a member of the object?

NOTE *The preceding ambiguous line will typically compile without any warnings or errors, but it will not produce the results that you are expecting.*

In order to disambiguate the names you can use the `this` pointer:

```
void SpreadsheetCell::setValue(double value)
{
    this->value = value;
    mString = doubleToString(this->value);
}
```

However, if you use the naming conventions described in Chapter 3, you will never encounter this type of name collision.

You can also use the `this` pointer to call a function or method that takes a pointer to an object from within a method of that object. For example, suppose you write a `printCell()` stand-alone function (not method) like this:

```
void printCell(const SpreadsheetCell& inCell)
{
    cout << inCell.getString() << endl;
}
```

If you want to call `printCell()` from the `setValue()` method, you must pass `*this` as the argument to give `printCell()` a reference to the `SpreadsheetCell` on which `setValue()` operates:

```
void SpreadsheetCell::setValue(double value)
{
    this->value = value;
    mString = doubleToString(this->value);
    printCell(*this);
}
```

NOTE Instead of writing a `printCell()` function, it would be more convenient to overload the `<<` operator, explained in Chapter 14. You can then use the following line to print a `SpreadsheetCell`:

```
cout << *this << endl;
```

Using Objects

The previous class definition says that a `SpreadsheetCell` consists of two member variables, four public methods, and two private methods. However, the class definition does not actually create any `SpreadsheetCells`; it just specifies their shape and behavior. In that sense, a class is similar to architectural blueprints. The blueprints specify what a house should look like, but drawing the blueprints doesn't build any houses. Houses must be constructed later based on the blueprints.

Similarly, in C++ you can construct a `SpreadsheetCell` “object” from the `SpreadsheetCell` class definition by declaring a variable of type `SpreadsheetCell`. Just as a builder can build more than one house based on a given set of blueprints, a programmer can create more than one `SpreadsheetCell` object from a `SpreadsheetCell` class. There are two ways to create and use objects: on the stack and on the heap, and both can be stored in arrays. Arrays are discussed later in this chapter.

Objects on the Stack

Here is some code that creates and uses `SpreadsheetCell` objects on the stack.

```
SpreadsheetCell myCell, anotherCell;
myCell.setValue(6);
anotherCell.setString("3.2");
cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

You create objects just as you declare simple variables, except that the variable type is the class name. The `.` in lines like `myCell.setValue(6);` is called the “dot” operator; it allows you to call methods on the object. If there were any public data members in the object, you could access them with the dot operator as well. Remember that public data members are not recommended.

The output of the program is:

```
cell 1: 6
cell 2: 3.2
```

Objects on the Heap

You can also dynamically allocate objects by using `new`:

```
SpreadsheetCell* myCellp = new SpreadsheetCell();
myCellp->setValue(3.7);
```

```

cout << "cell 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;
delete myCellp;
myCellp = nullptr;

```

When you create an object on the heap, you access its members through the “arrow” operator: `->`. The arrow combines dereferencing `(*)` and member access `(.)`. You could use those two operators instead, but doing so would be stylistically awkward:

```

SpreadsheetCell* myCellp = new SpreadsheetCell();
(*myCellp).setValue(3.7);
cout << "cell 1: " << (*myCellp).getValue() <<
    " " << (*myCellp).getString() << endl;
delete myCellp;
myCellp = nullptr;

```

Just as you must free other memory that you allocate on the heap, you must free the memory for objects that you allocate on the heap by calling `delete` on the objects. To avoid memory problems, it's highly recommended to use smart pointers as follows:

```

auto myCellp = make_unique<SpreadsheetCell>();
// Equivalent to:
// unique_ptr<SpreadsheetCell> myCellp(new SpreadsheetCell());
myCellp->setValue(3.7);
cout << "cell 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;

```

With smart pointers you don't need to manually free the memory, it will happen automatically.

WARNING *If you allocate an object with `new`, free it with `delete` when you are finished with it, or use smart pointers to manage the memory automatically.*

NOTE *If you don't use smart pointers, it is always a good idea to reset a pointer to the null pointer after deleting the object to which it pointed. You are not required to do this, but it will make debugging easier in case the pointer is accidentally used after deleting the object.*

OBJECT LIFE CYCLES

The object life cycle involves three activities: creation, destruction, and assignment. It is important to understand how and when objects are created, destroyed, and assigned, and how you can customize these behaviors.

Object Creation

Objects are created at the point you declare them (if they're on the stack) or when you explicitly allocate space for them with `new`, `new[]`, or a smart pointer. When an object is created, all its embedded objects are also created. For example:

```
#include <string>
class MyClass
{
    private:
        std::string mName;
};
int main()
{
    MyClass obj;
    return 0;
}
```

The embedded `string` object is created at the point where the `MyClass` object is created in the `main()` function and is destructed when its containing object is destructed.

It is often helpful to give variables initial values as you declare them. For example, you should usually initialize integer variables to 0 like this:

```
int x = 0;
```

Similarly, you should give initial values to objects. You can provide this functionality by declaring and writing a special method called a *constructor*, in which you can perform initialization work for the object. Whenever an object is created, one of its constructors is executed.

NOTE C++ programmers sometimes call a constructor a *ctor*.

Writing Constructors

Syntactically, a constructor is specified by a method name that is the same as the class name. A constructor never has a return type and may or may not have parameters. A constructor with no parameters is called the *default constructor*. There are certain contexts in which you may have to provide a default constructor and you will get compiler errors if you have not provided one. Default constructors are discussed later in the chapter.

Here is a first attempt at adding a constructor to the `SpreadsheetCell` class:

```
class SpreadsheetCell
{
    public:
        SpreadsheetCell(double initialValue);
        // Remainder of the class definition omitted for brevity
};
```

Just as you must provide implementations for normal methods, you must provide an implementation for the constructor:

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
{
    setValue(initialValue);
}
```

The `SpreadsheetCell` constructor is a member of the `SpreadsheetCell` class, so C++ requires the normal `SpreadsheetCell::` scope resolution before the constructor name. The constructor name itself is also `SpreadsheetCell`, so the code ends up with the funny looking `SpreadsheetCell::SpreadsheetCell`. The implementation simply makes a call to `setValue()` in order to set both the numeric and text representations.

Using Constructors

Using the constructor creates an object and initializes its values. You can use constructors with both stack-based and heap-based allocation.

Constructors on the Stack

When you allocate a `SpreadsheetCell` object on the stack, you use the constructor like this:

```
SpreadsheetCell myCell(5), anotherCell(4);
cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

Note that you do NOT call the `SpreadsheetCell` constructor explicitly. For example, do not use something like the following:

```
SpreadsheetCell myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

Similarly, you cannot call the constructor later. The following is also incorrect:

```
SpreadsheetCell myCell;
myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

Again, the only correct way to use the constructor on the stack is like this:

```
SpreadsheetCell myCell(5);
```

Constructors on the Heap

When you dynamically allocate a `SpreadsheetCell` object, you use the constructor like this:

```
auto smartCellp = make_unique<SpreadsheetCell>(4);
// ... do something with the cell, no need to delete the smart pointer

// Or with naked pointers, without smart pointers (not recommended)
SpreadsheetCell* myCellp = new SpreadsheetCell(5);
SpreadsheetCell* anotherCellp = nullptr;
```

```

anotherCellp = new SpreadsheetCell(4);
// ... do something with the cells
delete myCellp;           myCellp = nullptr;
delete anotherCellp;       anotherCellp = nullptr;

```

Note that you can declare a pointer to a `SpreadsheetCell` object without calling the constructor immediately, which is different from objects on the stack, where the constructor is called at the point of declaration.

Whenever you declare a pointer on the stack or in a class and don't immediately initialize the object, then it should be initialized to `nullptr` like in the previous declaration for `anotherCellp`. If you don't assign it to `nullptr`, the pointer is undefined. Accidentally using an undefined pointer will cause unexpected and difficult-to-diagnose memory corruption. If you initialize it to `nullptr`, using that pointer will cause a memory access error in most operating environments, instead of producing unexpected results.

Remember to call `delete` on objects that you dynamically allocate with `new` or use smart pointers!

Providing Multiple Constructors

You can provide more than one constructor in a class. All constructors have the same name (the name of the class), but different constructors must take a different number of arguments or different argument types. In C++, if you have more than one function with the same name, the compiler will select the one whose parameter types match the types at the call site. This is called *overloading* and is discussed in detail in Chapter 8.

In the `SpreadsheetCell` class, it is helpful to have two constructors: one to take an initial `double` value and one to take an initial `string` value. Here is the new class definition:

```

class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const std::string& initialValue);
    // Remainder of the class definition omitted for brevity
};

```

Here is the implementation of the second constructor:

```

SpreadsheetCell::SpreadsheetCell(const string& initialValue)
{
    setString(initialValue);
}

```

And here is some code that uses the two different constructors:

```

SpreadsheetCell aThirdCell("test"); // Uses string-arg ctor
SpreadsheetCell aFourthCell(4.4);   // Uses double-arg ctor
auto aThirdCellp = make_unique<SpreadsheetCell>("4.4"); // string-arg ctor
cout << "aThirdCell: " << aThirdCell.getValue() << endl;
cout << "aFourthCell: " << aFourthCell.getValue() << endl;
cout << "aThirdCellp: " << aThirdCellp->getValue() << endl;

```

When you have multiple constructors, it is tempting to attempt to implement one constructor in terms of another. For example, you might want to call the double constructor from the string constructor as follows:

```
SpreadsheetCell::SpreadsheetCell(const string& initialValue)
{
    SpreadsheetCell(stringToDouble(initialValue));
}
```

That seems to make sense. After all, you can call normal class methods from within other methods. The code will compile, link, and run, **but will not do what you expect**. The explicit call to the `SpreadsheetCell` constructor actually creates a new temporary unnamed object of type `SpreadsheetCell`. It does not call the constructor for the object that you are supposed to be initializing.

However, C++ supports *delegating constructors* which allow you to call other constructors from the same class from inside the ctor-initializer. This is discussed later in this chapter.

Default Constructors

A *default constructor* is a constructor that requires no arguments. It is also called a *0-argument constructor*. With a default constructor, you can give initial values to data members even though the client did not specify them.

When You Need a Default Constructor

Consider arrays of objects. The act of creating an array of objects accomplishes two tasks: It allocates contiguous memory space for all the objects and it calls the default constructor on each object. C++ fails to provide any syntax to tell the array creation code directly to call a different constructor. For example, if you do not define a default constructor for the `SpreadsheetCell` class, the following code does not compile:

```
SpreadsheetCell cells[3]; // FAILS compilation without default constructor
SpreadsheetCell* myCellp = new SpreadsheetCell[10]; // Also FAILS
```

You can circumvent this restriction for stack-based arrays by using *initializers* like these:

```
SpreadsheetCell cells[3] = {SpreadsheetCell(0), SpreadsheetCell(23),
                           SpreadsheetCell(41)};
```

However, it is usually easier to ensure that your class has a default constructor if you intend to create arrays of objects of that class. If you haven't defined your own constructors, the compiler will automatically create a default constructor for you. This compiler-generated constructor is discussed in a next section.

A default constructor is also required for classes that you want to store in an STL container like `std::vector`.

Default constructors are also useful when you want to create objects of that class inside other classes, which is shown later in this chapter under the section *Constructor Initializers*.

How To Write a Default Constructor

Here is part of the `SpreadsheetCell` class definition with a default constructor:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    // Remainder of the class definition omitted for brevity
};
```

Here is a first crack at an implementation of the default constructor. There is no need for the constructor to initialize `mString` to the empty string, because the default constructor of `std::string` is automatically called and initializes `mString` to the empty string.

```
SpreadsheetCell::SpreadsheetCell()
{
    mValue = 0;
}
```

You use the default constructor on the stack like this:

```
SpreadsheetCell myCell;
myCell.setValue(6);
cout << "cell 1: " << myCell.getValue() << endl;
```

The preceding code creates a new `SpreadsheetCell` called `myCell`, sets its value, and prints out its value. Unlike other constructors for stack-based objects, you do not call the default constructor with function-call syntax. Based on the syntax for other constructors, you might be tempted to call the default constructor like this:

```
SpreadsheetCell myCell(); // WRONG, but will compile.
myCell.setValue(6);      // However, this line will not compile.
cout << "cell 1: " << myCell.getValue() << endl;
```

Unfortunately, the line attempting to call the default constructor will compile. The line following it will not compile. The problem is that your compiler thinks the first line is actually a function declaration for a function with the name `myCell` that takes zero arguments and returns a `SpreadsheetCell` object. When it gets to the second line, it thinks that you're trying to use a function name as an object!

WARNING When creating an object on the stack, omit parentheses for the default constructor.

For heap-based object allocation, the default constructor can be used as follows:

```
auto smartCellp = make_unique<SpreadsheetCell>();
// Or with a naked pointer (not recommended)
SpreadsheetCell* myCellp = new SpreadsheetCell();
// Or
// SpreadsheetCell* myCellp = new SpreadsheetCell;
// ... use myCellp
delete myCellp;    myCellp = nullptr;
```

Compiler-Generated Default Constructor

The first `SpreadsheetCell` class definition in this chapter looked as follows:

```
class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue() const;
private:
    double mValue;
};
```

This definition does not declare a default constructor, but still, the code that follows works perfectly.

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

The following definition is the same as the preceding definition except that it adds an explicit constructor, accepting a `double`. It still does not explicitly declare a default constructor.

```
class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue); // No default constructor
    // Remainder of the class definition omitted for brevity
};
```

With this definition, the following code will not compile anymore:

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

What's going on here? The reason is that if you don't specify *any* constructors, the compiler will write one for you that doesn't take any arguments. This *compiler-generated default constructor* calls the default constructor on all object members of the class, but does not initialize the language primitives such as `int` and `double`. Nonetheless, it allows you to create objects of that class. However, if you declare a default constructor, or any other constructor, the compiler no longer generates a default constructor for you.

NOTE A *default constructor* is the same thing as a 0-argument constructor. The term *default constructor* does not refer only to the constructor automatically generated if you fail to declare any constructors. It refers to the constructor which is defaulted to if no arguments are required.

Explicitly Defaulted Constructors

In C++03 or older if your class required a number of explicit constructors accepting arguments but also a default constructor that does nothing, you had to explicitly write your empty default constructor as follows:

```
class MyClass
{
public:
    MyClass() {}
    MyClass(int i);
};
```

However, it's recommended that interface files contain only declarations of public methods without any implementations. The preceding class definition violates this. The solution was to define the class as follows:

```
class MyClass
{
public:
    MyClass();
    MyClass(int i);
};
```

The implementation of the empty default constructor in the implementation file would be:

```
MyClass::MyClass() {}
```

To avoid having to write empty default constructors manually, C++ now supports the concept of *explicitly defaulted constructors*. This allows you to write the class definition as follows without the need to implement the default constructor in the implementation file.

```
class MyClass
{
public:
    MyClass() = default;
    MyClass(int i);
};
```

MyClass defines a custom constructor that accepts one integer. However, the compiler will still generate a standard compiler generated default constructor due to the use of the `default` keyword.

Explicitly Deleted Constructors

C++ also supports the concept of *explicitly deleted constructors*. For example, you can define a class with only static methods for which you do not want to write any constructors and you also do not want the compiler to generate the default constructor. In that case you need to explicitly delete the default constructor:

```
class MyClass
{
public:
    MyClass() = delete;
};
```

Constructor Initializers

Up to now, this chapter initialized data members in the body of a constructor, for example:

```
SpreadsheetCell::SpreadsheetCell()
{
    mValue = 0;
}
```

C++ provides an alternative method for initializing data members in the constructor, called the *constructor initializer* or *ctor-initializer*. Here is the 0-argument `SpreadsheetCell` constructor rewritten to use the ctor-initializer syntax:

```
SpreadsheetCell::SpreadsheetCell() : mValue(0)
{
}
```

As you can see, the ctor-initializer appears syntactically between the constructor argument list and the opening brace for the body of the constructor. The list starts with a colon and is separated by commas. Each element in the list is an initialization of a data member using function notation, or a call to a base class constructor (see Chapter 9), or a call to a delegated constructor discussed later.

Initializing data members with a ctor-initializer provides different behavior than does initializing data members inside the constructor body itself. When C++ creates an object, it must create all the data members of the object before calling the constructor. As part of creating these data members, it must call a constructor on any of them that are themselves objects. By the time you assign a value to an object inside your constructor body, you are not actually constructing that object. You are only modifying its value. A ctor-initializer allows you to provide initial values for data members as they are created, which is more efficient than assigning values to them later. The default initialization for strings gives them the empty string; so explicitly initializing `mString` to the empty string is superfluous.

If your class has as data member an object without a default constructor, you have to use the ctor-initializer to properly construct that object. For example, take the following `SpreadsheetCell` class:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell(double d);
};
```

The implementation of the constructor is as follows:

```
SpreadsheetCell::SpreadsheetCell(double d) { }
```

This class only has one explicit constructor accepting a `double` and does not include a default constructor. You can use this class as a data member of another class as follows:

```
class SomeClass
{
public:
    SomeClass();
private:
    SpreadsheetCell mCell;
};
```

And implement the `SomeClass` constructor as follows:

```
SomeClass::SomeClass() { }
```

However, with this implementation, the code will not compile. The compiler does not know how to initialize the `mCell` data member of `SomeClass` because it does not have a default constructor.

The solution is to initialize the `mCell` data member in the ctor-initializer as follows:

```
SomeClass::SomeClass() : mCell(1.0) { }
```

NOTE *Ctor-initializers allow initialization of data members at the time of their creation.*

Some programmers prefer to assign initial values in the body of the constructor. However, several data types must be initialized in a ctor-initializer. The following table summarizes them:

DATA TYPE	EXPLANATION
const data members	You cannot legally assign a value to a <code>const</code> variable after it is created. Any value must be supplied at the time of creation.
Reference data members	References cannot exist without referring to <i>something</i> .
Object data members for which there is no default constructor	C++ attempts to initialize member objects using a default constructor. If no default constructor exists, it cannot initialize the object.
Base classes without default constructors	[Covered in Chapter 9]

There is one important caveat with ctor-initializers: They initialize data members in the order that they appear in the class definition, not their order in the ctor-initializer. Take the following definition for the `SpreadsheetCell` class:

```
class SpreadsheetCell
{
public:
    // Code omitted for brevity
private:
    // Code omitted for brevity
    double mValue;
    std::string mString;
};
```

Suppose you write your `SpreadsheetCell` string constructor to use a ctor-initializer like this:

```
SpreadsheetCell::SpreadsheetCell(const string& initialValue) :
    mString(initialValue), mValue(stringToDouble(mString)) // INCORRECT ORDER!
{}
```

The code will compile (although some compilers issue a warning), but the program does not work correctly. You might assume that `mString` will be initialized before `mValue` because `mString` is listed first in the ctor-initializer. But C++ doesn't work that way. The `SpreadsheetCell` class declares `mValue` before `mString`; thus, the ctor-initializer tries to initialize `mValue` before `mString`. However, the code to initialize `mValue` tries to use the value of `mString`, which is not yet initialized! The solution in this case is to use the `initialValue` argument instead of `mString` when initializing `mValue`. You should also swap their order in the ctor-initializer to avoid confusion:

```
SpreadsheetCell::SpreadsheetCell(const string& initialValue) :
    mValue(stringToDouble(initialValue)), mString(initialValue)
{}
```

WARNING *Ctor-Initializers initialize data members in their declared order in the class definition, not their order in the ctor-initializer list.*

Copy Constructors

There is a special constructor in C++ called a *copy constructor* that allows you to create an object that is an exact copy of another object. If you don't write a copy constructor yourself, C++ generates one for you that initializes each data member in the new object from its equivalent data member in the source object. For object data members, this initialization means that their copy constructors are called.

Here is the declaration for a copy constructor in the `SpreadsheetCell` class:

```
class SpreadsheetCell
{
```

```

public:
    SpreadsheetCell(const SpreadsheetCell& src);
    // Remainder of the class definition omitted for brevity
};

```

The copy constructor takes a `const` reference to the source object. Like other constructors, it does not return a value. Inside the constructor, you should copy all the data fields from the source object. Technically, of course, you can do whatever you want in the copy constructor, but it's generally a good idea to follow expected behavior and initialize the new object to be a copy of the old one. Here is a sample implementation of the `SpreadsheetCell` copy constructor:

```

SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src) :
    mValue(src.mValue), mString(src.mString)
{
}

```

Note the use of the ctor-initializer.

NOTE *Given a set of member variables, called `m1`, `m2`, ... `mn`, the compiler-generated copy constructor can be expressed as:*

```

classname::classname(const classname& src) :
    m1(src.m1), m2(src.m2), ... mn(src.mn) { }

```

Therefore, in most circumstances, there is no need for you to specify a copy constructor yourself. However, under certain conditions, this default copy constructor is not sufficient. These conditions are covered in Chapter 8.

When the Copy Constructor Is Called

The default semantics for passing arguments to functions in C++ is pass-by-value. That means that the function or method receives a copy of the value or object. Thus, whenever you pass an object to a function or method the compiler calls the copy constructor of the new object to initialize it. For example, suppose that the definition of the `setString()` method in the `SpreadsheetCell` class is as follows:

```

void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = string.ToDouble(mString);
}

```

Recall that the C++ `string` is actually a class, not a built-in type. When your code makes a call to `setString()` passing a `string` argument, the `string` parameter `inString` is initialized with a call to its copy constructor. The argument to the copy constructor is the `string` you passed to `setString()`. In the following example, the `string` copy constructor is executed for the `inString` object in `setString()` with `name` as its parameter.

```
SpreadsheetCell myCell;
string name = "heading one";
myCell.setString(name); // Copies name
```

When the `setString()` method finishes, `inString` is destroyed. Because it was only a copy of `name`, `name` remains intact.

The copy constructor is also called whenever you return an object from a function or method. In this case, the compiler creates a temporary, unnamed object through its copy constructor. Chapter 25 explores the impact of temporary objects in more detail. You can avoid the overhead of copy constructors by passing parameters as `const` references, as has been done throughout this chapter.

Calling the Copy Constructor Explicitly

You can use the copy constructor explicitly as well. It is often useful to be able to construct one object as an exact copy of another. For example, you might want to create a copy of a `SpreadsheetCell` object like this:

```
SpreadsheetCell myCell2(4);
SpreadsheetCell myCell3(myCell2); // myCell3 has the same values as myCell2
```

Passing Objects by Reference

In order to avoid copying objects when you pass them to functions and methods you can declare that the function or method takes a *reference* to the object. Passing objects by reference is usually more efficient than passing them by value, because only the address of the object is copied, not the entire contents of the object. Additionally, pass-by-reference avoids problems with dynamic memory allocation in objects, which is discussed in Chapter 8.

When you pass an object by reference, the function or method using the object reference could change the original object. When you're only using pass-by-reference for efficiency, you should preclude this possibility by declaring the object `const` as well. This is known as passing objects by `const` reference and has been done in all previous code examples. For example, all methods of the `SpreadsheetCell` class that require a string are declared with a `const string` reference parameter. The class definition would look as follows if you would declare those methods without reference parameters.

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(std::string initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    void setValue(double inValue);
    double getValue() const;
    void setString(std::string inString);
    std::string getString() const;
private:
    std::string doubleToString(double inValue) const;
    double stringToDouble(std::string inString) const;
```

```

        double mValue;
        std::string mString;
    };

```

Here is the implementation for `setString()`. Note that the method body remains the same; only the parameter type is different.

```

void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = string.ToDouble(mString);
}

```

NOTE *For performance reasons, it is best to pass objects by const reference instead of by value.*

The `doubleToString()` method of the `SpreadsheetCell` class always returns a `string` by value because the implementation of the method creates a local `string` object which at the end of the method is returned to the caller. Returning a reference to this `string` wouldn't work because the `string` to which it references will be destroyed when the function exits.

Explicitly Defaulted and Deleted Copy Constructor

You can explicitly default or delete a compiler generated copy constructor as follows:

```
SpreadsheetCell(const SpreadsheetCell& src) = default;
```

or

```
SpreadsheetCell(const SpreadsheetCell& src) = delete;
```

Initializer-List Constructors

An *initializer-list constructor* is a constructor with `std::initializer_list<T>` as first argument, without any additional arguments or with additional arguments having default values. Before you can use the `std::initializer_list<T>` template you need to include the `<initializer_list>` header. The following class demonstrates its use.

```

class EvenSequence
{
public:
    EvenSequence(initializer_list<double> args)
    {
        if (args.size() % 2 != 0) {
            throw invalid_argument("initializer_list should "
                "contain even number of elements.");
        }
        mSequence.reserve(args.size());
    }
}

```

```
        for (auto value : args) {
            mSequence.push_back(value);
        }
    }
    void dump() const
    {
        for (auto value : mSequence) {
            cout << value << ", ";
        }
        cout << endl;
    }
private:
    vector<double> mSequence;
};
```

Inside the initializer-list constructor you can access the elements of the initializer-list with a range-based `for` loop. You can get the number of elements in the initializer-list with the `size()` method.

The `EvenSequence` initializer-list constructor uses a range-based `for` loop to copy elements from the given `initializer_list`. In practice, it's recommended to use STL algorithms as much as possible. STL algorithms are discussed in detail in Chapter 17. As an example, the previous range-based `for` loop with the `push_back()` call can be rewritten using an STL algorithm as follows:

```
mSequence.insert(cend(mSequence), cbegin(args), cend(args));
```

Objects of `EvenSequence` can be constructed as follows:

```
EvenSequence p1 = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
p1.dump();
try {
    EvenSequence p2 = {1.0, 2.0, 3.0};
} catch (const invalid_argument& e) {
    cout << e.what() << endl;
}
```

The construction of `p2` will throw an exception because it has an odd number of elements in the initializer-list. The preceding equal signs are optional and can be left out, for example:

```
EvenSequence p1{1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
```

The STL has full support for initializer-list constructors. For example, the `std::vector` container can be initialized by using an initializer-list:

```
std::vector<std::string> myVec = {"String 1", "String 2", "String 3"};
```

Without initializer-list constructors, one way to initialize the `vector` is by using several `push_back()` calls:

```
std::vector<std::string> myVec;
myVec.push_back("String 1");
myVec.push_back("String 2");
myVec.push_back("String 3");
```

Initializer lists are not limited to constructors and can also be used with normal functions as explained in Chapter 10.

In-Class Member Initializers

Since C++11, member variables can be initialized directly in the class definition. For example:

```
#include <string>
class MyClass
{
    private:
        int mInt = 1;
        std::string mStr = "test";
};
```

The only way you could initialize `mInt` and `mStr` before C++11 was within a constructor body, or with a ctor-initializer as follows:

```
#include <string>
class MyClass
{
    public:
        MyClass() : mInt(1), mStr("test") {}
    private:
        int mInt;
        std::string mStr;
};
```

Before C++11, only `static const integral` member variables could be initialized in the class definition. For example:

```
#include <string>
class MyClass
{
    private:
        static const int kI1 = 1;    // OK
        static const std::string kStr = "test"; // Error: not integral type
        static int sI2 = 2;          // Error: not const
        const int kI3 = 3;          // Error: not static
};
```

Delegating Constructors

Delegating constructors allow constructors to call another constructor from the same class. However, this call cannot be placed in the constructor body; it must be in the constructor initializer and it must be the only member-initializer in the list. Following is an example:

```
SpreadsheetCell::SpreadsheetCell(const string& initialValue)
    : SpreadsheetCell(stringtoDouble(initialValue))
{}
```

When this string constructor (the delegating constructor) is called, it will first delegate the call to the target, double constructor. When the target constructor returns, the body of the delegating constructor will be executed.

Make sure you avoid constructor recursion while using delegate constructors. For example:

```
class MyClass
{
    MyClass(char c) : MyClass(1.2) { }
    MyClass(double d) : MyClass('m') { }
};
```

The first constructor will delegate to the second constructor, which delegates back to the first one. The behavior of such code is undefined by the standard and depends on the compiler.

Summary of Compiler-Generated Constructors

The compiler automatically generates a 0-argument and copy constructor for every class. However, the constructors you define yourself replace these according to the following rules:

IF YOU DEFINE THEN THE COMPILER GENERATES AND YOU CAN CREATE AN OBJECT ...
[no constructors]	A 0-argument constructor A copy constructor	With no arguments: <code>SpreadsheetCell cell;</code> As a copy of another object: <code>SpreadsheetCell myCell(cell);</code>
A 0-argument constructor only	A copy constructor	With no arguments: <code>SpreadsheetCell cell;</code> As a copy of another object: <code>SpreadsheetCell myCell(cell);</code>
A copy constructor only	No constructors	Theoretically, as a copy of another object. Practically, you can't create any objects.
A single-argument or multi-argument non-copy constructor only	A copy constructor	With arguments: <code>SpreadsheetCell cell(6);</code> As a copy of another object: <code>SpreadsheetCell myCell(cell);</code>
A 0-argument constructor as well as a single-argument or multi-argument non-copy constructor	A copy constructor	With no arguments: <code>SpreadsheetCell cell;</code> With arguments: <code>SpreadsheetCell myCell(5);</code> As a copy of another object: <code>SpreadsheetCell anotherCell(cell);</code>

Note the lack of symmetry between the default constructor and the copy constructor. As long as you don't define a copy constructor explicitly, the compiler creates one for you. On the other hand, as soon as you define *any* constructor, the compiler stops generating a default constructor.

As mentioned before in this chapter, the automatic generation of a default constructor and a default copy constructor can be influenced by defining them as explicitly defaulted or explicitly deleted.

NOTE *A final type of constructor is called a move constructor, required to implement move semantics. Move semantics can be used to increase performance in certain situations and is discussed in detail in Chapter 10.*

Object Destruction

When an object is destroyed, two events occur: The object's *destructor* method is called, and the memory it was taking up is freed. The destructor is your chance to perform any cleanup work for the object, such as freeing dynamically allocated memory or closing file handles. If you don't declare a destructor, the compiler will write one for you that does recursive memberwise destruction and allows the object to be deleted. The section on dynamic memory allocation in Chapter 8 shows you how to write a destructor.

Objects on the stack are destroyed when they go *out of scope*, which means whenever the current function, method, or other execution *block* ends. In other words, whenever the code encounters an ending curly brace, any objects created on the stack within those curly braces are destroyed. The following program shows this behavior:

```
int main()
{
    SpreadsheetCell myCell(5);
    if (myCell.getValue() == 5) {
        SpreadsheetCell anotherCell(6);
    } // anotherCell is destroyed as this block ends.
    cout << "myCell: " << myCell.getValue() << endl;
    return 0;
} // myCell is destroyed as this block ends.
```

Objects on the stack are destroyed in the reverse order of their declaration (and construction). For example, in the following code fragment, `myCell2` is allocated before `anotherCell2`, so `anotherCell2` is destroyed before `myCell2` (note that you can start a new code block at any point in your program with an opening curly brace):

```
{
    SpreadsheetCell myCell2(4);
    SpreadsheetCell anotherCell2(5); // myCell2 constructed before anotherCell2
} // anotherCell2 destroyed before myCell2
```

This ordering applies to objects that are data members of other objects. Recall that data members are initialized in the order of their declaration in the class. Thus, following the rule that objects are

destroyed in the reverse order of their construction, data member objects are destroyed in the reverse order of their declaration in the class.

Objects allocated on the heap without the help of a smart pointer are not destroyed automatically. You must call `delete` on the object pointer to call its destructor and free the memory. The following program shows this behavior:

```
int main()
{
    SpreadsheetCell* cellPtr1 = new SpreadsheetCell(5);
    SpreadsheetCell* cellPtr2 = new SpreadsheetCell(6);
    cout << "cellPtr1: " << cellPtr1->getValue() << endl;
    delete cellPtr1; // Destroys cellPtr1
    cellPtr1 = nullptr;
    return 0;
} // cellPtr2 is NOT destroyed because delete was not called on it.
```

WARNING *Do not write programs like the preceding example where `cellPtr2` was not deleted. Make sure you always free dynamically allocated memory by calling `delete` or `delete[]` depending on whether the memory was allocated using `new` or `new[]` or better yet, use smart pointers as discussed earlier.*

NOTE *There are tools that are able to detect unfreed objects. These tools are discussed in Chapter 22.*

Assigning to Objects

Just as you can assign the value of one `int` to another in C++, you can assign the value of one object to another. For example, the following code assigns the value of `myCell` to `anotherCell`:

```
SpreadsheetCell myCell(5), anotherCell;
anotherCell = myCell;
```

You might be tempted to say that `myCell` is “copied” to `anotherCell`. However, in the world of C++, “copying” only occurs when an object is being initialized. If an object already has a value that is being overwritten, the more accurate term is “assigned” to. Note that the facility that C++ provides for copying is the copy constructor. Since it is a constructor, it can only be used for object creation, not for later assignments to the object.

Therefore, C++ provides another method in every class to perform assignment. This method is called the *assignment operator*. Its name is `operator=` because it is actually an overloading of the `=` operator for that class. In the preceding example, the assignment operator for `anotherCell` is called, with `myCell` as the argument.

NOTE The assignment operator as explained in this section is sometimes called the copy assignment operator because both the left-hand side and the right-hand side object stay alive after the assignment. This distinction is made because there is also a move assignment operator in which the right-hand side object will be destroyed after the assignment for performance reasons. This move assignment operator is explained in Chapter 10.

As usual, if you don't write your own assignment operator, C++ writes one for you to allow objects to be assigned to one another. The default C++ assignment behavior is almost identical to its default copying behavior: It recursively assigns each data member from the source to the destination object.

Declaring an Assignment Operator

Here is another attempt at the `SpreadsheetCell` class definition, this time including an assignment operator:

```
class SpreadsheetCell
{
public:
    // Remainder of the class definition omitted for brevity
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    // Remainder of the class definition omitted for brevity
};
```

The assignment operator, like the copy constructor, takes a `const` reference to the source object. In this case, the source object is called `rhs` which stands for “right-hand side” of the equals sign. The object on which the assignment operator is called is the left-hand side of the equals sign.

Unlike a copy constructor, the assignment operator returns a reference to a `SpreadsheetCell` object. The reason is that assignments can be *chained*, as in the following example:

```
myCell = anotherCell = aThirdCell;
```

When that line is executed, the first thing that happens is that the assignment operator for `anotherCell` is called with `aThirdCell` as its “right-hand side” parameter. Next, the assignment operator for `myCell` is called. However, its parameter is not `anotherCell`. Its right-hand side is the *result* of the assignment of `aThirdCell` to `anotherCell`. If that assignment fails to return a result, there is nothing to pass to `myCell`.

You might be wondering why the assignment operator for `myCell` can't just take `anotherCell`. The reason is that using the equals sign is actually just shorthand for what is really a method call. When you look at the line in its full functional syntax, you can see the problem:

```
myCell.operator=(anotherCell.operator=(aThirdCell));
```

Now, you can see that the `operator=` call from `anotherCell` must return a value, which is passed to the `operator=` call for `myCell`. The correct value to return is `anotherCell` itself, so it can serve

as the source for the assignment to `myCell`. However, returning `anotherCell` directly would be inefficient, so you can return a reference to `anotherCell`.

WARNING *You could actually declare the assignment operator to return whatever type you wanted, including void. However, you should always return a reference to the object on which it is called because that's what clients expect.*

Defining an Assignment Operator

The implementation of the assignment operator is similar to that of a copy constructor, with several important differences. First, a copy constructor is called only for initialization, so the destination object does not yet have valid values. An assignment operator can overwrite the current values in an object. This consideration doesn't really come into play until you have dynamically allocated memory in your objects. See Chapter 8 for details.

Second, it's legal in C++ to assign an object to itself. For example, the following code compiles and runs:

```
SpreadsheetCell cell(4);
cell = cell; // Self-assignment
```

Your assignment operator shouldn't prohibit self-assignment, but also shouldn't perform a full assignment if it happens. Thus, assignment operators should check for self-assignment at the beginning of the method and return immediately.

Here is the definition of the assignment operator for the `SpreadsheetCell` class:

```
SpreadsheetCell& SpreadsheetCell::operator=(const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
```

The previous line checks for self-assignment, but is a bit cryptic. Self-assignment occurs when the left-hand side and the right-hand side of the equals sign are the same. One way to tell if two objects are the same is if they occupy the same memory location — more explicitly, if pointers to them are equal. Recall that `this` is a pointer to an object accessible from any method called on the object. Thus, `this` is a pointer to the left-hand side object. Similarly, `&rhs` is a pointer to the right-hand side object. If these pointers are equal, the assignment must be self-assignment, but because the return type is `SpreadsheetCell&` a correct value must be returned. All assignment operators return `*this`, and the self-assignment case is no exception:

```
    return *this;
}
```

`this` is a pointer to the object on which the method executes, so `*this` is the object itself. The compiler will return a reference to the object to match the declared return value. Now, if it is not self-assignment, you have to do an assignment to every member:

```
mValue = rhs.mValue;
mString = rhs.mString;
```

Here the method copies the values.

```
    return *this;
}
```

Finally it returns `*this`, as explained previously.

Explicitly Defaulted and Deleted Assignment Operator

You can explicitly default or delete a compiler generated assignment operator as follows:

```
SpreadsheetCell& operator=(const SpreadsheetCell& rhs) = default;
```

or

```
SpreadsheetCell& operator=(const SpreadsheetCell& rhs) = delete;
```

Distinguishing Copying from Assignment

It is sometimes difficult to tell when objects are initialized with a copy constructor rather than assigned to with the assignment operator. Essentially, things that look like a declaration are going to be using copy constructors and things that look like assignment statements will be handled by the assignment operator. Consider the following code:

```
SpreadsheetCell myCell(5);
SpreadsheetCell anotherCell(myCell);
```

`AnotherCell` is constructed with the copy constructor.

```
SpreadsheetCell aThirdCell = myCell;
```

`aThirdCell` is also constructed with the copy constructor, because this is a declaration. This line does not call `operator=!` This syntax is just another way to write:

`SpreadsheetCell aThirdCell(myCell);`. However:

```
anotherCell = myCell; // Calls operator= for anotherCell.
```

Here `anotherCell` has already been constructed, so the compiler calls `operator=`.

Objects as Return Values

When you return objects from functions or methods, it is sometimes difficult to see exactly what copying and assignment is happening. Suppose that the code for `getString()` looks like this:

```
string SpreadsheetCell::getString() const
{
    return mString;
}
```

Now consider the following code:

```
SpreadsheetCell myCell2(5);
string s1;
s1 = myCell2.getString();
```

When `getString()` returns `mString`, the compiler actually creates an unnamed temporary `string` object by calling a `string` copy constructor. When you assign this result to `s1`, the assignment operator is called for `s1` with the temporary `string` as a parameter. Then, the temporary `string` object is destroyed. Thus, the single line of code invokes the copy constructor and the assignment operator (for two different objects). However, compilers are free to implement Return Value Optimization (RVO) to optimize this process.

In case you're not confused enough, consider this code:

```
SpreadsheetCell myCell3(5);
string s2 = myCell3.getString();
```

In this case, `getString()` still creates a temporary unnamed `string` object when it returns `mString`. But now `s2` gets its copy constructor called, not its assignment operator.

With *move semantics*, the compiler can use a *move constructor* instead of a copy constructor to return `mString` from `getString()`. This is more efficient. Move semantics is discussed in Chapter 10.

If you ever forget the order in which these things happen or which constructor or operator is called, you can easily figure it out by temporarily including helpful output in your code or by stepping through it with a debugger.

Copy Constructors and Object Members

You should also note the difference between assignment and copy constructor calls in constructors. If an object contains other objects, the compiler-generated copy constructor calls the copy constructors of each of the contained objects recursively. When you write your own copy constructor, you can provide the same semantics by using a `ctor-initializer`, as shown previously. If you omit a data member from the `ctor-initializer`, the compiler performs default initialization on it (a call to the 0-argument constructor for objects) before executing your code in the body of the constructor. Thus, by the time the body of the constructor executes, all object data members have already been initialized.

For example, you could write your copy constructor like this:

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
: mString(src.mString)
{
    mValue = src.mValue;
}
```

However, when you assign values to data members in the body of the copy constructor, you are using the assignment operator on them, not the copy constructor, because they have already been initialized, as described previously.

In this example, `mString` is initialized using the copy constructor, while `mValue` is assigned to using the assignment operator.

SUMMARY

This chapter covered the fundamental aspects of C++’s facilities for object-oriented programming: classes and objects. It first reviewed the basic syntax for writing classes and using objects, including access control. Then, it covered object life cycles: when objects are constructed, destructed, and assigned, and what methods those actions invoke. The chapter included details of the constructor syntax, including ctor-initializers and initializer-list constructors, and introduced the notion of copy assignment operators. It also specified exactly which constructors the compiler writes for you, and under what circumstances, and explained that default constructors require no arguments.

For some of you, this chapter was mostly review. For others, it hopefully opened your eyes to the world of object-oriented programming in C++. In any case, now that you are proficient with objects and classes, read Chapter 8 to learn more about their tricks and subtleties.

8

Mastering Classes and Objects

WHAT'S IN THIS CHAPTER?

- How to use dynamic memory allocation in objects
- The different kinds of data members you can have (*static, const, reference*)
- The different kinds of methods you can implement (*static, const, inline*)
- The details of method overloading
- How to work with default parameters
- How to use nested classes
- How to make classes friends of other classes
- What operator overloading is
- How to write separate interface and implementation classes.

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Chapter 7 started the discussion on classes and objects. Now it's time to master their subtleties so you can use them to their full potential. By reading this chapter, you will learn how to manipulate and exploit some of the most powerful aspects of the C++ language in order to write safe, effective, and useful classes.

Many of the concepts in this chapter arise in advanced C++ programming, especially in the standard template library.

DYNAMIC MEMORY ALLOCATION IN OBJECTS

Sometimes you don't know how much memory you will need before your program actually runs. As you know, the solution is to dynamically allocate as much space as you need during program execution. Classes are no exception. Sometimes you don't know how much memory an object will need when you write the class. In that case, the object should dynamically allocate memory. Dynamically allocated memory in objects provides several challenges, including freeing the memory, handling object copying, and handling object assignment.

The Spreadsheet Class

Chapter 7 introduced the `SpreadsheetCell` class. This chapter moves on to write the `Spreadsheet` class. As with the `SpreadsheetCell` class, the `Spreadsheet` class will evolve throughout this chapter. Thus, the various attempts do not always illustrate the best way to do every aspect of class writing. To start, a `Spreadsheet` is simply a two-dimensional array of `SpreadsheetCells`, with methods to set and retrieve cells at specific locations in the `Spreadsheet`. Although most spreadsheet applications use letters in one direction and numbers in the other to refer to cells, this `Spreadsheet` uses numbers in both directions. Here is a first attempt at a class definition for a simple `Spreadsheet` class:

```
#include "SpreadsheetCell.h"
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    void setCellAt(int x, int y, const SpreadsheetCell& cell);
    SpreadsheetCell& getCellAt(int x, int y);
private:
    bool inRange(int val, int upper);
    int mWidth, mHeight;
    SpreadsheetCell** mCells;
};
```

NOTE The `Spreadsheet` class uses normal pointers for the `mCells` array. This is done throughout this chapter to show the consequences and to explain how you should handle dynamic memory in classes. In production code, you should use one of the standard C++ containers, like `std::vector` which greatly simplifies the implementation of `Spreadsheet`, but then you won't learn how to correctly handle dynamic memory using naked pointers. In modern C++ you should never use naked pointers, but you might come across it in existing code in which case you need to know how it works.

Note that the `Spreadsheet` class does not contain a standard two-dimensional array of `SpreadsheetCells`. Instead, it contains a `SpreadsheetCell**`. The reason is that each `Spreadsheet` object might have different dimensions, so the constructor of the class must dynamically allocate the two-dimensional array based on the client-specified height and width. In order to allocate

dynamically a two-dimensional array you need to write the following code. Note that in C++, unlike in Java, it's not possible to simply write new `SpreadsheetCell[mWidth] [mHeight]`.

```
#include "Spreadsheet.h"
Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
}
```

The resultant memory for a `Spreadsheet` called `s1` on the stack with width four and height three is shown in Figure 8-1.

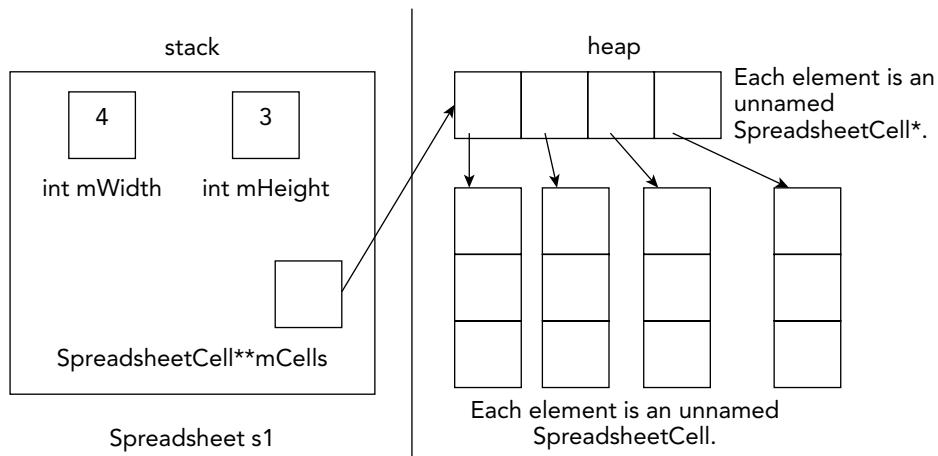


FIGURE 8-1

The implementations of the set and retrieval methods are straightforward:

```
void Spreadsheet::setCellAt(int x, int y, const SpreadsheetCell& cell)
{
    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        throw std::out_of_range("");
    }
    mCells[x][y] = cell;
}
SpreadsheetCell& Spreadsheet::getCellAt(int x, int y)
{
    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        throw std::out_of_range("");
    }
    return mCells[x][y];
}
```

Note that these two methods use a helper method `inRange()` to check that `x` and `y` represent valid coordinates in the spreadsheet. Attempting to access an array element at an out-of-range index will cause the program to malfunction. This example uses exceptions which are mentioned in Chapter 1 and described in detail in Chapter 13.

Freeing Memory with Destructors

Whenever you are finished with dynamically allocated memory, you should free it. If you dynamically allocate memory in an object, the place to free that memory is in the destructor. The compiler guarantees that the destructor will be called when the object is destroyed. Here is the `Spreadsheet` class definition from earlier with a destructor:

```
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    ~Spreadsheet();
    // Code omitted for brevity
};
```

The destructor has the same name as the name of the class (and of the constructors), preceded by a tilde (~). The destructor takes no arguments, and there can only be one of them.

Here is the implementation of the `Spreadsheet` class destructor:

```
Spreadsheet::~Spreadsheet()
{
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;
    mCells = nullptr;
}
```

This destructor frees the memory that was allocated in the constructor. However, no rule requires you to free memory in the destructor. You can write whatever code you want in the destructor, but it is a good idea to use it only for freeing memory or disposing of other resources.

Handling Copying and Assignment

Recall from Chapter 7 that, if you don't write a copy constructor and an assignment operator yourself, C++ writes them for you. These compiler-generated methods recursively call the copy constructor or assignment operator on object data members. However, for primitives, such as `int`, `double`, and pointers, they provide *shallow* or *bitwise* copying or assignment: They just copy or assign the data members from the source object directly to the destination object.

That presents problems when you dynamically allocate memory in your object. For example, the following code copies the spreadsheet `s1` to initialize `s` when `s1` is passed to the `printSpreadsheet()` function.

```

#include "Spreadsheet.h"
void printSpreadsheet(Spreadsheet s)
{
    // Code omitted for brevity.
}
int main()
{
    Spreadsheet s1(4, 3);
    printSpreadsheet(s1);
    return 0;
}

```

The `Spreadsheet` contains one pointer variable: `mCells`. A shallow copy of a spreadsheet gives the destination object a copy of the `mCells` pointer, but not a copy of the underlying data. Thus, you end up with a situation where both `s` and `s1` have a pointer to the same data, as shown in Figure 8-2.

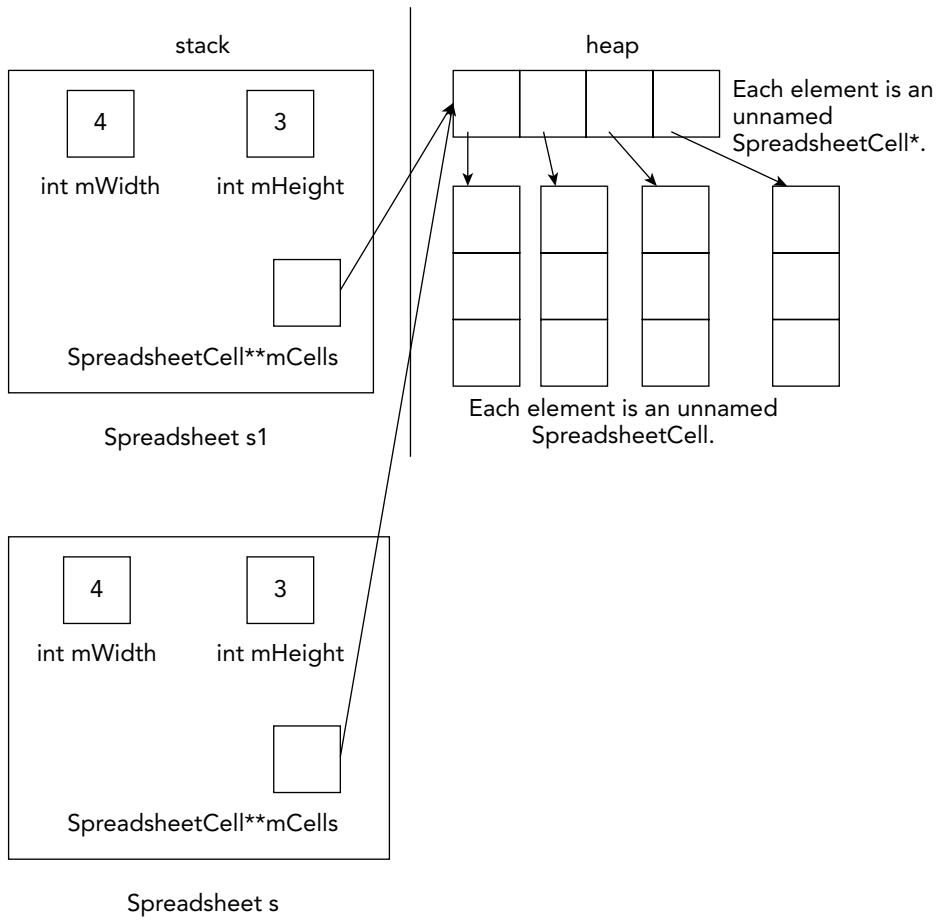


FIGURE 8-2

If `s` were to change something to which `mCells` points, that change would show up in `s1` too. Even worse, when the `printSpreadsheet()` function exits, `s`'s destructor is called, which frees the memory pointed to by `mCells`. That leaves the situation shown in Figure 8-3.

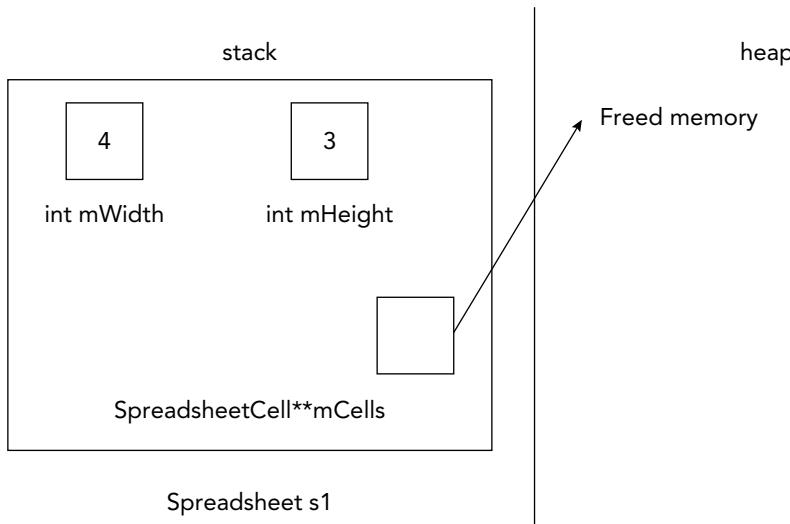


FIGURE 8-3

Now `s1` has a pointer which no longer points to valid memory. This is called a *dangling pointer*.

Unbelievably, the problem is even worse with assignment. Suppose that you had the following code:

```
Spreadsheets s1(2, 2), s2(4, 3);
s1 = s2;
```

After both objects are constructed, you would have the memory layout shown in Figure 8-4.

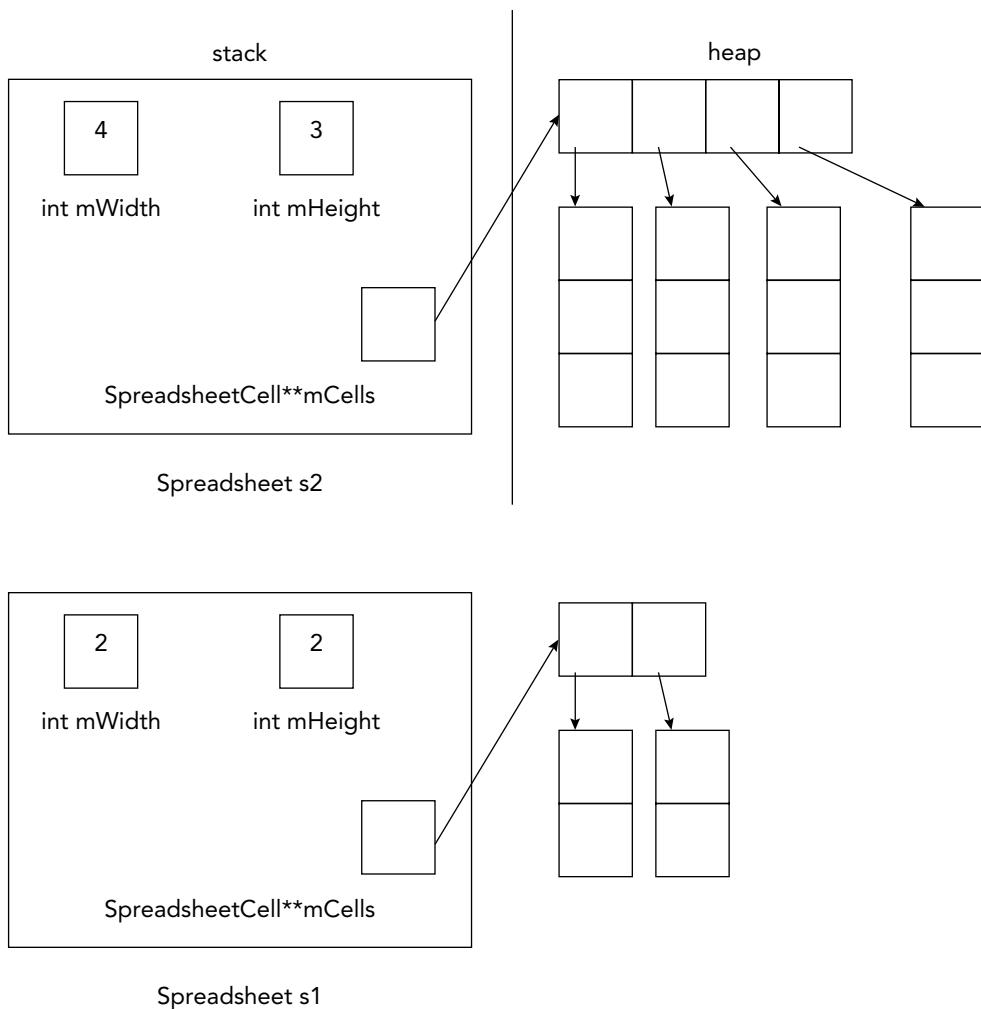


FIGURE 8-4

After the assignment statement, you would have the layout shown in Figure 8-5.

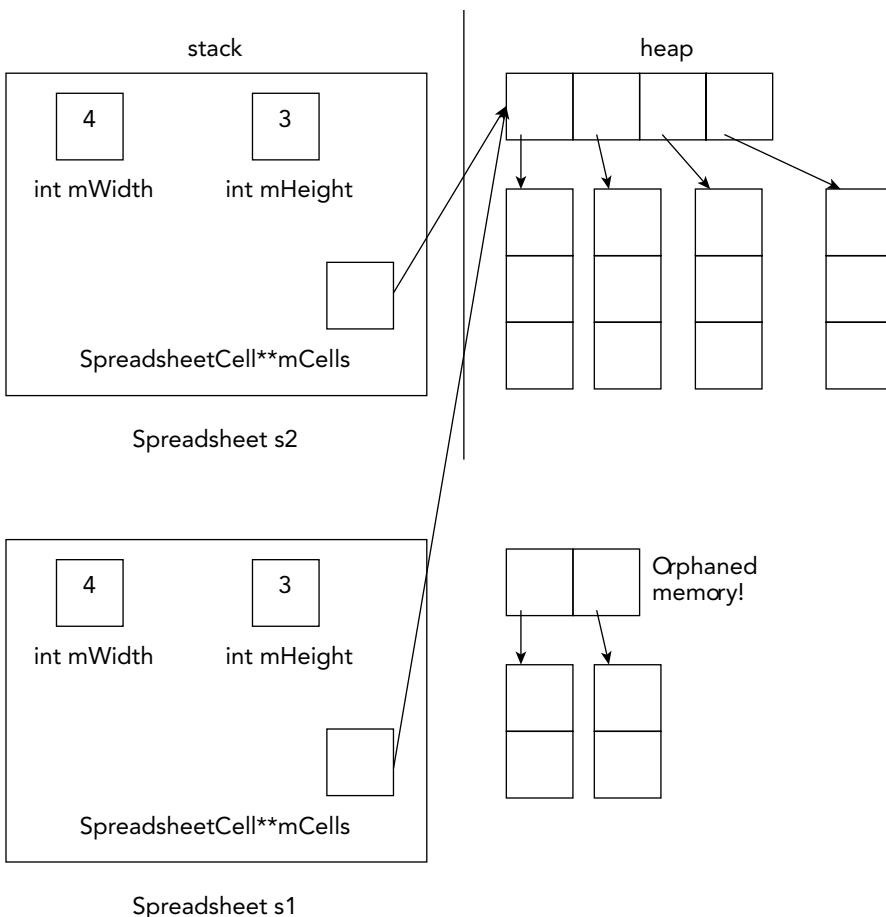


FIGURE 8-5

Now, not only do the `mCells`s pointers in `s1` and `s2` point to the same memory, but also you have *orphaned* the memory to which `mCells` in `s1` previously pointed. This is called a memory leak. That is why in assignment operators you must first free the memory referenced by the left-hand side, and then do a deep copy.

As you can see, relying on C++'s default copy constructor or assignment operator is not always a good idea.

WARNING Whenever you have dynamically allocated memory in a class, you should write your own copy constructor and assignment operator to provide a deep copy of the memory.

The Spreadsheet Copy Constructor

Here is a declaration for a copy constructor in the `Spreadsheet` class:

```
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    Spreadsheet(const Spreadsheet& src);
    // Code omitted for brevity
};
```

Here is a first definition of the copy constructor:

```
Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
    for (int i = 0; i < mWidth; i++) {
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}
```

Note that the copy constructor copies all data members, including `mWidth` and `mHeight`, not just the pointer data members. The rest of the code in the copy constructor provides a deep copy of the `mCells` dynamically allocated two-dimensional array. There is no need to delete the existing `mCells` because this is a copy constructor and therefore there is no existing `mCells` yet in this object.

The Spreadsheet Assignment Operator

Here is the definition for the `Spreadsheet` class with an assignment operator:

```
class Spreadsheet
{
public:
    // Code omitted for brevity
    Spreadsheet& operator=(const Spreadsheet& rhs);
    // Code omitted for brevity
};
```

Here is the implementation of the assignment operator for the `Spreadsheet` class, with explanations interspersed. Note that when an object is assigned to, it already has been initialized. Thus, you must free any dynamically allocated memory before allocating new memory. You can think of an assignment operator as a combination of a destructor and a copy constructor. You are essentially “reincarnating” the object with new life (or data) when you assign to it.

The first line of code in any assignment operator checks for self-assignment.

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return *this;
    }
}
```

This self-assignment check is required, not only for efficiency, but also for correctness. If the preceding self-assignment test was removed, the code will not work correctly. The assignment operator would first delete `mCells` and allocate a new `mCells` for the left-hand side object. However, both the left-hand side object and the right-hand side object are the same, so by deleting and re-allocating `mCells` for the left-hand side, you are doing the same for the right-hand side. The cells that you wanted to copy will be lost.

Because this is an assignment operator, the object being assigned to already has `mCells` initialized. You need to free these cells up.

```
// Free the old memory.
for (int i = 0; i < mWidth; i++) {
    delete [] mCells[i];
}
delete [] mCells;
mCells = nullptr;
```

You must free all the memory before reallocating it, or you will create a memory leak. This chunk of code is identical to the destructor, so you should make a separate method to cleanup the memory and call it both from the destructor and from the assignment operator. The next step is to copy the memory.

```
// Copy the new memory.
mWidth = rhs.mWidth;
mHeight = rhs.mHeight;
mCells = new SpreadsheetCell* [mWidth];
for (int i = 0; i < mWidth; i++) {
    mCells[i] = new SpreadsheetCell[mHeight];
}
for (int i = 0; i < mWidth; i++) {
    for (int j = 0; j < mHeight; j++) {
        mCells[i][j] = rhs.mCells[i][j];
    }
}
return *this;
}
```

Note that this code looks remarkably like the code in the copy constructor. The following section explains how you can avoid this code duplication.

The assignment operator completes the “big 3” routines for managing dynamically allocated memory in an object: the **destructor**, the **copy constructor**, and the **assignment operator**. Whenever you find yourself writing one of those methods you should write all of them.

WARNING Whenever a class dynamically allocates memory, write a destructor, copy constructor, and assignment operator.

NOTE Next to copying, C++ also supports move semantics, which requires a move constructor and move assignment operator. These can be used to increase performance in certain situations and are discussed in detail in Chapter 10.

Common Helper Routines for Copy Constructor and Assignment Operator

The copy constructor and the assignment operator are quite similar. Thus, it's usually convenient to factor the common tasks into a helper method. For example, you could add a `copyFrom()` method to the `Spreadsheet` class, and rewrite the copy constructor and assignment operator to use it like this:

```
void Spreadsheet::copyFrom(const Spreadsheet& src)
{
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
    for (int i = 0; i < mWidth; i++) {
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}
Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    copyFrom(src);
}
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return *this;
    }
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;
    mCells = nullptr;
    // Copy the new memory.
    copyFrom(rhs);
    return *this;
}
```

Disallowing Assignment and Pass-By-Value

Sometimes when you dynamically allocate memory in your class, it's easiest just to prevent anyone from copying or assigning to your objects. You can do this by explicitly deleting your `operator=` and copy constructor. That way, if anyone tries to pass the object by value, return it from a function or method, or assign to it, the compiler will complain. Here is a `Spreadsheet` class definition that prevents assignment and pass-by-value:

```
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    Spreadsheet(const Spreadsheet& src) = delete;
    ~Spreadsheet();
    Spreadsheet& operator=(const Spreadsheet& rhs) = delete;
// Code omitted for brevity
};
```

You don't provide implementations for deleted copy constructors and assignment operators. The linker will never look for them because the compiler won't allow code to call them. When you write code to copy or assign to a `Spreadsheet` object, the compiler will complain with a message like:

```
'Spreadsheet &Spreadsheet::operator =(const Spreadsheet &)' : attempting to
reference a deleted function
```

NOTE *If your compiler doesn't support explicitly deleting member functions, then you can disallow copying and assigning by making your copy constructor and assignment operator private without any implementation.*

DIFFERENT KINDS OF DATA MEMBERS

C++ gives you many choices for data members. In addition to declaring simple data members in your classes, you can create static data members that all objects of the class share, `const` members, reference members, `const` reference members, and more. This section explains the intricacies of these different kinds of data members.

static Data Members

Sometimes giving each object of a class a copy of a variable is overkill or won't work. The data member might be specific to the class, but not appropriate for each object to have its own copy. For example, you might want to give each spreadsheet a unique numerical identifier. You would need a counter that starts at 0 from which each new object could obtain its ID. This spreadsheet counter really belongs to the `Spreadsheet` class, but it doesn't make sense for each `Spreadsheet` object to have a copy of it because you would have to keep all the counters synchronized somehow.

C++ provides a solution with *static data members*. A static data member is a data member associated with a class instead of an object. You can think of static data members as global variables specific to a class. Here is the `Spreadsheet` class definition, including the new `static` counter data member:

```
class Spreadsheet
{
    // Omitted for brevity
private:
    static int sCounter;
};
```

In addition to listing static class members in the class definition, you will have to allocate space for them in a source file, usually the source file in which you place your class method definitions. You can initialize them at the same time, but note that unlike normal variables and data members, they are initialized to 0 by default. `static` pointers are initialized to `nullptr`. Here is the code to allocate space for and initialize the `sCounter` member:

```
int Spreadsheet::sCounter;
```

This code appears outside of any function or method bodies. It's almost like declaring a global variable, except that the `Spreadsheet::` scope resolution specifies that it's part of the `Spreadsheet` class.

Accessing static Data Members within Class Methods

You can use static data members as if they were regular data members from within class methods. For example, you might want to create an `mId` member of the `Spreadsheet` class and initialize it from the `sCounter` member in the `Spreadsheet` constructor. Here is the `Spreadsheet` class definition with an `mId` member:

```
class Spreadsheet
{
public:
    // Omitted for brevity
    int getId() const;
private:
    // Omitted for brevity
    static int sCounter;
    int mId;
};
```

Here is an implementation of the `Spreadsheet` constructor that assigns the initial ID:

```
Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mId = sCounter++;
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
}
```

As you can see, the constructor can access `sCounter` as if it were a normal member. Remember to assign an ID in the copy constructor as well:

```
Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    mId = sCounter++;
    copyFrom(src);
}
```

You should not copy the ID in the assignment operator. Once an ID is assigned to an object it should never change. Thus it's recommended to make `mId` a `const` data member. `Const` data members are discussed in the next section.

Accessing static Data Members Outside Methods

Access control specifiers apply to `static` data members: `sCounter` is `private`, so it cannot be accessed from outside class methods. If `sCounter` was `public`, you could access it from outside class methods by specifying that the variable is part of the `Spreadsheet` class with the `::` scope resolution operator:

```
int c = Spreadsheet::sCounter;
```

However, it's not recommended to have `public` data members. You should grant access through `public` get/set methods. If you want to grant access to a `static` data member, you need to implement `static` get/set methods, which are explained later in this chapter.

const Data Members

Data members in your class can be declared `const`, meaning they can't be changed after they are created and initialized. You should use `static const` data members in place of global constants when the constants apply only to the class. For example, you might want to specify a maximum height and width for spreadsheets. If the user tries to construct a spreadsheet with a greater height or width than the maximum, the maximum is used instead. You can make the max height and width `static const` members of the `Spreadsheet` class:

```
class Spreadsheet
{
public:
    // Omitted for brevity
    static const int kMaxHeight = 100;
    static const int kMaxWidth = 100;
};
```

You can use these new constants in your constructor as shown in the following section of code (note the use of the ternary operator):

```
Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth < kMaxWidth ? inWidth : kMaxWidth),
    mHeight(inHeight < kMaxHeight ? inHeight : kMaxHeight)
{
```

```
    mId = sCounter++;
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
}
```

NOTE Instead of automatically clamping the width and height to their maximum, you could also decide to throw an exception when the width or height exceed their maximum. However, the destructor *will not* be called when you throw an exception from a constructor. So, you need to be careful with this. Details are explained in Chapter 13.

NOTE Non-static data members can also be declared `const`. For example, the `mId` data member could be declared as `const`. Since you cannot assign to a `const` data member, you need to initialize them in your ctor-initializers.

`kMaxHeight` and `kMaxWidth` are public, so you can access them from anywhere in your program as if they were global variables, but with slightly different syntax: You must specify that the variable is part of the `Spreadsheet` class with the `::` scope resolution operator:

```
cout << "Maximum height is: " << Spreadsheet::kMaxHeight << endl;
```

Reference Data Members

Spreadsheets and SpreadsheetCells are great, but they don't make a very useful application by themselves. You need code to control the whole spreadsheet program, which you could package into a SpreadsheetApplication class.

The implementation of this class is unimportant at the moment. For now, consider this architecture problem: How can spreadsheets communicate with the application? The application stores a list of spreadsheets, so it can communicate with the spreadsheets. Similarly, each spreadsheet could store a reference to the application object. The `Spreadsheet` class must then know about the `SpreadsheetApplication` class and the `SpreadsheetApplication` class must know about the `Spreadsheet` class. This is a circular reference and cannot be solved with normal `#includes`. The solution is to use a *forward declaration* in one of the header files. Here is the new `Spreadsheet` class definition that uses a forward declaration to tell the compiler about the `SpreadsheetApplication` class:

```
class SpreadsheetApplication; // forward declaration
class Spreadsheet
{
    public:
```

```
    Spreadsheet(int inWidth, int inHeight,
                SpreadsheetApplication& theApp);
    // Code omitted for brevity.
private:
    // Code omitted for brevity.
    SpreadsheetApplication& mTheApp;
};
```

This definition adds a `SpreadsheetApplication` reference as a data member. It's recommended to use a reference in this case instead of a pointer because a `Spreadsheet` should always refer to a `SpreadsheetApplication`. This would not be guaranteed with a pointer.

Note that storing a reference to the application is only done to demonstrate the use of references as data members. It's not recommended to couple the `Spreadsheet` and `SpreadsheetApplication` classes together in this way, but instead to use the MVC (Model-View-Controller) paradigm.

The application reference is given to each `Spreadsheet` in its constructor. A reference cannot exist without referring to something, so `mTheApp` must be given a value in the ctor-initializer of the constructor:

```
Spreadsheet::Spreadsheet(int inWidth, int inHeight,
                         SpreadsheetApplication& theApp)
: mWidth(inWidth < kMaxWidth ? inWidth : kMaxWidth),
  mHeight(inHeight < kMaxHeight ? inHeight : kMaxHeight), mTheApp(theApp)
{
    // Code omitted for brevity.
}
```

You must also initialize the reference member in the copy constructor:

```
Spreadsheet::Spreadsheet(const Spreadsheet& src) :
    mTheApp(src.mTheApp)
{
    mId = sCounter++;
    copyFrom(src);
}
```

Remember that after you have initialized a reference you cannot change the object to which it refers. It's not possible to assign to references in the assignment operator.

const Reference Data Members

Your reference members can refer to `const` objects just as normal references can refer to `const` objects. For example, you might decide that `Spreadsheets` should only have a `const` reference to the application object. You can simply change the class definition to declare `mTheApp` as a `const` reference:

```
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight,
```

```

        const SpreadsheetApplication& theApp);
        // Code omitted for brevity.
private:
    // Code omitted for brevity.
    const SpreadsheetApplication& mTheApp;
};

```

There is an important difference between using a `const` reference versus a non-`const` reference. The `const` reference `SpreadsheetApplication` data member can only be used to call `const` methods on the `SpreadsheetApplication` object. If you try to call a non-`const` method through a `const` reference, you will get a compiler error.

It's also possible to have a `static` reference member or a `static const` reference member, but you will rarely find the need for something like that.

MORE ABOUT METHODS

C++ also provides myriad choices for methods. This section explains all the tricky details.

static Methods

Methods, like data members, sometimes apply to the class as a whole, not to each object. You can write `static` methods as well as data members. As an example, consider the `SpreadsheetCell` class from Chapter 7. It has two helper methods: `string.ToDouble()` and `doubleToString()`. These methods don't access information about specific objects, so they could be `static`. Here is the class definition with these methods `static`:

```

class SpreadsheetCell
{
    // Omitted for brevity
private:
    static std::string doubleToString(double val);
    static double stringToDouble(const std::string& str);
    // Omitted for brevity
};

```

The implementations of these two methods are identical to the previous implementations. You don't repeat the `static` keyword in front of the method definitions. However, note that `static` methods are not called on a specific object, so they have no `this` pointer, and are not executing for a specific object with access to its non-`static` members. In fact, a `static` method is just like a regular function. The only difference is that it can access `private` and `protected` `static` data members of the class. It can also access `private` and `protected` non-`static` data members on objects of the same type, if those objects are made visible to the `static` method, for example by passing in a reference or pointer to such an object.

You call a `static` method just like a regular function from within any method of the class. Thus, the implementation of all methods in `SpreadsheetCell` can stay the same. Outside of the class, you need to qualify the method name with the class name using the scope resolution operator (as for `static` members). Access control applies as usual.

You might want to make `stringToDouble()` and `doubleToString()` `public` so that other code outside the class could make use of them. If so, you could call them from anywhere like this:

```
string str = SpreadsheetCell::doubleToString(5.0);
```

const Methods

A `const` object is an object whose value cannot be changed. If you have a `const`, reference to `const` or pointer to `const` object, the compiler will not let you call any methods on that object unless those methods guarantee that they won't change any data members. The way you guarantee that a method won't change data members is to mark the method itself with the `const` keyword. Here is the `SpreadsheetCell` class with the methods that don't change any data member marked `const`:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    double getValue() const;
    const std::string& getString() const;
    // Omitted for brevity
};
```

The `const` specification is part of the method prototype and must accompany its definition as well:

```
double SpreadsheetCell::getValue() const
{
    return mValue;
}
const std::string& SpreadsheetCell::getString() const
{
    return mString;
}
```

Marking a method as `const` signs a contract with client code guaranteeing that you will not try to change the internal values of the object within the method. If you try to declare a method `const` that actually modifies a data member, the compiler will complain. You also cannot declare a `static` method, such as the `doubleToString()` and `stringToDouble()` methods from the previous section `const` because it is redundant. Static methods do not have an instance of the `class` so it would be impossible for them to change internal values. `const` works by making it appear inside the method that you have a `const` reference to each data member. Thus, if you try to change the data member the compiler will flag an error.

A non-`const` object can call `const` and non-`const` methods. However, a `const` object can only call `const` methods. Here are some examples:

```
SpreadsheetCell myCell(5);
cout << myCell.getValue() << endl;           // OK
myCell.setString("6");                         // OK
```

```
const SpreadsheetCell& anotherCell = myCell;
cout << anotherCell.getValue() << endl; // OK
anotherCell.setString("6");           // Compilation Error!
```

You should get into the habit of declaring `const` all methods that don't modify the object so that you can use references to `const` objects in your program.

Note that `const` objects can still be destroyed, and their destructor can be called. Nevertheless, destructors are not allowed to be declared `const`.

mutable Data Members

Sometimes you write a method that is “logically” `const` but happens to change a data member of the object. This modification has no effect on any user-visible data, but is technically a change, so the compiler won’t let you declare the method `const`. For example, suppose that you want to profile your spreadsheet application to obtain info about how often data is being read. A crude way to do this would be to add a counter to the `SpreadsheetCell` class that counts each call to `getValue()` or `getString()`. Unfortunately, that makes those methods non-`const` in the compiler’s eyes, which is not what you intended. The solution is to make your new counter variable `mutable`, which tells the compiler that it’s okay to change it in a `const` method. Here is the new `SpreadsheetCell` class definition:

```
class SpreadsheetCell
{
    // Omitted for brevity
private:
    double mValue;
    std::string mString;
    mutable int mNumAccesses = 0;
};
```

Here are the definitions for `getValue()` and `getString()`:

```
double SpreadsheetCell::getValue() const
{
    mNumAccesses++;
    return mValue;
}
const std::string& SpreadsheetCell::getString() const
{
    mNumAccesses++;
    return mString;
}
```

Method Overloading

You’ve already noticed that you can write multiple constructors in a class, all of which have the same name. These constructors differ only in the number or types of their parameters. You can do the same thing for any method or function in C++. Specifically, you can *overload* the function or method name by using it for multiple functions, as long as the number or types of the parameters

differ. For example, in the `spreadsheetCell` class you could rename both `setString()` and `setValue()` to `set()`. The class definition now looks like this:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    void set(double inValue);
    void set(const std::string& inString);
    // Omitted for brevity
};
```

The implementations of the `set()` methods stay the same. Note that the double constructor that previously called `setValue()` must now call `set()`. When you write code to call `set()`, the compiler determines which instance to call based on the parameter you pass: If you pass a `string` the compiler calls the `string` instance; if you pass a `double` the compiler calls the `double` instance. This is called *overload resolution*.

You might be tempted to do the same thing for `getValue()` and `getString()`: Rename each of them to `get()`. However, that does not compile. C++ does not allow you to overload a method name based only on the return type of the method because in many cases it would be impossible for the compiler to determine which instance of the method to call. For example, if the return value of the method is not captured anywhere, the compiler has no way to tell which instance of the method you wanted.

Note also that you can overload a method based on `const`. That is, you can write two methods with the same name and same parameters, one of which is declared `const` and one of which is not. The compiler will call the `const` method if you have a `const` object and the non-`const` method if you have a non-`const` object.

Overloaded methods can be explicitly deleted, which can be used to disallow calling a member function with particular arguments. For example, suppose you have the following class:

```
class MyClass
{
public:
    void foo(int i);
};
```

The `foo()` method can be called as follows:

```
MyClass c;
c.foo(123);
c.foo(1.23);
```

For the third line, the compiler will convert the `double` value (1.23) to an integer value (1) and then call `foo(int i)`. The compiler might give you a warning, but it will perform this implicit conversion. You can prevent the compiler from performing this conversion by explicitly deleting a `double` instance of `foo()`:

```
class MyClass
{
public:
```

```

    void foo(int i);
    void foo(double d) = delete;
};

```

With this change, an attempt to call `foo()` with a `double` will be flagged as an error by the compiler, instead of performing a conversion to an integer.

Default Parameters

A feature similar to method overloading in C++ is *default parameters*. You can specify defaults for function and method parameters in the prototype. If the user specifies those arguments, the defaults are ignored. If the user omits those arguments, the default values are used. There is a limitation, though: You can only provide defaults for a continuous list of parameters starting from the *rightmost parameter*. Otherwise, the compiler would not be able to match missing arguments to default parameters. Default parameters can be used in functions, methods, and constructors. For example, you can assign default values to the width and height in your `Spreadsheet` constructor:

```

class Spreadsheet
{
public:
    Spreadsheet(const SpreadsheetApplication& theApp,
                int inWidth = kMaxWidth, int inHeight = kMaxHeight);
    // Omitted for brevity
};

```

The implementation of the `Spreadsheet` constructor stays the same. Note that you specify the default parameters only in the method declaration, but not in the definition.

Now you can call the `Spreadsheet` constructor with one, two, or three arguments even though there is only one non-copy constructor:

```

SpreadsheetApplication theApp;
Spreadsheet s1(theApp);
Spreadsheet s2(theApp, 5);
Spreadsheet s3(theApp, 5, 6);

```

A constructor with defaults for all its parameters can function as a default constructor. That is, you can construct an object of that class without specifying any arguments. If you try to declare both a default constructor and a multi-argument constructor with defaults for all its parameters, the compiler will complain because it won't know which constructor to call if you don't specify any arguments.

Note that anything you can do with default parameters you can do with method overloading. You could write three different constructors, each of which takes a different number of parameters. However, default parameters allow you to write only one constructor to take three different numbers of arguments. You should use the mechanism with which you are most comfortable.

Inline Methods

C++ gives you the ability to recommend that a call to a method or function should not actually be implemented in the generated code as a call to a separate block of code. Instead, the compiler should insert the method or function body directly into the code where the method or function call is made.

This process is called *inlining*, and methods or functions that want this behavior are called `inline` methods or functions. Inlining is safer than using `#define` macros.

You can specify an `inline` method or function by placing the `inline` keyword in front of its name in the function or method definition. For example, you might want to make the accessor methods of the `SpreadsheetCell` class `inline`, in which case you would define them like this:

```
inline double SpreadsheetCell::getValue() const
{
    mNumAccesses++;
    return mValue;
}
inline const std::string& SpreadsheetCell::getString() const
{
    mNumAccesses++;
    return mString;
}
```

This gives a hint to the compiler to replace calls to `getValue()` and `getString()` with the actual method body instead of generating code to make a function call. Note that the `inline` keyword is just a hint for the compiler. The compiler can ignore it if it thinks it would hurt performance.

There is one caveat: Definitions of `inline` methods and functions must be available in every source file in which they are called. That makes sense if you think about it: How can the compiler substitute the function body if it can't see the function definition? Thus, if you write `inline` functions or methods you should place the definitions in a header file along with their prototypes. For methods, this means placing the definitions in the `.h` file that includes the class definition.

NOTE Advanced C++ compilers do not require you to put definitions of `inline` methods in a header file. For example, Microsoft Visual C++ supports Link-Time Code Generation (LTCG) which will automatically inline small function bodies, even if they are not declared as `inline` and even if they are not defined in a header file. When you use such a compiler, make use of it, and don't put the definitions in the header file. This way, your interface files stay clean without any implementation details visible in it.

C++ provides an alternate syntax for declaring `inline` methods that doesn't use the `inline` keyword at all. Instead, you place the method definition directly in the class definition. Here is a `SpreadsheetCell` class definition with this syntax:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    double getValue() const { mNumAccesses++; return mValue; }
    const std::string& getString() const { mNumAccesses++; return mString; }
    // Omitted for brevity
};
```

NOTE If you single-step with a debugger on a function call that is inlined, some advanced C++ debuggers will jump to the actual source code of the inline function, giving you the illusion of a function call, but in reality, the code is inlined.

Many C++ programmers discover the `inline` method syntax and employ it without understanding the ramifications of marking a method `inline`. Marking a method or function as `inline` only gives a hint to the compiler to inline it. Compilers will only inline the simplest methods and functions. If you define an `inline` method that the compiler doesn't want to inline, it will silently ignore the directive.

Modern compilers will take metrics like code bloat into account before deciding to inline a method or function, and they will not inline anything that is not cost-effective.

NESTED CLASSES

Class definitions can contain more than just member functions and data members. You can also write nested classes and structs, declare `typedefs`, or create enumerated types. Anything declared inside a class is in the scope of that class. If it is `public`, you can access it outside the class by scoping it with the `ClassName::` scope resolution syntax.

You can provide a class definition inside another class definition. For example, you might decide that the `SpreadsheetCell` class is really part of the `Spreadsheet` class. You could define both of them like this:

```
class Spreadsheet
{
public:
    class SpreadsheetCell
    {
public:
        SpreadsheetCell();
        SpreadsheetCell(double initialValue);
        // Omitted for brevity
    private:
        double mValue;
        std::string mString;
        mutable int mNumAccesses = 0;
    };
    Spreadsheet(const SpreadsheetApplication& theApp,
               int inWidth = kMaxWidth, int inHeight = kMaxHeight);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();
    // Remainder of Spreadsheet declarations omitted for brevity
};
```

Now, the `SpreadsheetCell` class is defined inside the `Spreadsheet` class, so anywhere you refer to a `SpreadsheetCell` outside of the `Spreadsheet` class you must qualify the name with the

`Spreadsheet`:: scope. This applies even to the method definitions. For example, the default constructor now looks like this:

```
Spreadsheet::SpreadsheetCell::SpreadsheetCell() : mValue(0)
{
}
```

This syntax can quickly become clumsy. For example, the definition of the `SpreadsheetCell` assignment operator now looks like this:

```
Spreadsheet::SpreadsheetCell& Spreadsheet::SpreadsheetCell::operator=
    const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
        return *this;
    }
    mValue = rhs.mValue;
    mString = rhs.mString;
    mNumAccesses = rhs.mNumAccesses;
    return *this;
}
```

In fact, you must even use the syntax for return types (but not parameters) of methods in the `Spreadsheet` class itself:

```
Spreadsheet::SpreadsheetCell& Spreadsheet::getCellAt(int x, int y)
{
    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        throw std::out_of_range("");
    }
    return mCells[x][y];
}
```

You can avoid the clumsy syntax by using a type alias to rename `Spreadsheet::SpreadsheetCell` to something more manageable like `SCell`:

```
using SCell = Spreadsheet::SpreadsheetCell;
```

This type alias should go outside the `Spreadsheet` class definition, or else you will have to qualify the type alias name itself with `Spreadsheet::` to get `Spreadsheet::SCell`. That wouldn't do you much good!

Now you can write the default constructor like this:

```
SCell::SpreadsheetCell() : mValue(0)
{
}
```

Normal access control applies to nested class definitions. If you declare a `private` or `protected` nested class, you can only use it inside the outer class.

You should generally use nested class definitions only for trivial classes. It is really too clumsy for something like the `SpreadsheetCell` class.

ENUMERATED TYPES INSIDE CLASSES

If you want to define a number of constants inside a class, you should use an enumerated type instead of a collection of `#defines`. For example, you can add support for cell coloring to the `SpreadsheetCell` class as follows:

```
class SpreadsheetCell
{
    public:
        // Omitted for brevity
        enum class Colors { Red = 1, Green, Blue, Yellow };
        void setColor(Colors color);
    private:
        // Omitted for brevity
        Colors mColor = Colors::Red;
};
```

The implementation of the `setColor()` method is straightforward:

```
void SpreadsheetCell::setColor(Colors color)
{
    mColor = color;
}
```

The new method can be used as follows:

```
SpreadsheetCell myCell(5);
myCell.setColor(SpreadsheetCell::Colors::Blue);
```

Using an enumerated type is the preferred solution instead of using `#defines` as follows:

```
#define SPREADSHEETCELL_RED 1
#define SPREADSHEETCELL_GREEN 2
#define SPREADSHEETCELL_BLUE 3
#define SPREADSHEETCELL_YELLOW 4
class SpreadsheetCell
{
    public:
        // Omitted for brevity
        void setColor(int color);
    private:
        // Omitted for brevity
        int mColor;
};
```

When you use `#defines`, you have to use an integer parameter for the `setColor()` function instead of a clear type like the `Colors` enumerated type.

FRIENDS

C++ allows classes to declare that other classes, or member functions of other classes, or nonmember functions are *friends*, and can access protected and private data members and methods. For example, the `SpreadsheetCell` class could specify that the `Spreadsheet` class is its “friend” like this:

```
class SpreadsheetCell
{
public:
    friend class Spreadsheet;
    // Remainder of the class omitted for brevity
};
```

Now all the methods of the `Spreadsheet` class can access the private and protected data members and methods of the `SpreadsheetCell` class.

If you only want to make a specific member function of the `Spreadsheet` class a friend, you can do that as follows:

```
class SpreadsheetCell
{
public:
    friend void Spreadsheet::setCellAt(int x, int y,
                                         const SpreadsheetCell& cell);
    // Remainder of the class omitted for brevity
};
```

Note that a class needs to know which other classes, methods, or functions wish to be its friends; a class, method or function cannot declare itself to be a friend of some other class and access the non-public names of that class.

You might, for example, want to write a function to verify that the string of a `SpreadsheetCell` object is not empty. You might want this verification routine to be outside the `SpreadsheetCell` class to model an external audit, but the function should be able to access the internal data members of the object in order to check it properly. Here is the `SpreadsheetCell` class definition with a `friend checkSpreadsheetCell()` function:

```
class SpreadsheetCell
{
public:
    friend bool checkSpreadsheetCell(const SpreadsheetCell& cell);
    // Omitted for brevity
};
```

The `friend` declaration in the class serves as the function’s prototype. There’s no need to write the prototype elsewhere (although it’s harmless to do so).

Here is the function definition:

```
bool checkSpreadsheetCell(const SpreadsheetCell& cell)
{
    return !(cell.mString.empty());
}
```

You write this function just like any other function, except that you can directly access private and protected data members of the `SpreadsheetCell` class. You don't repeat the `friend` keyword on the function definition.

`friend` classes and methods are easy to abuse; they allow you to violate the principle of encapsulation by exposing internals of your class to other classes or functions. Thus, you should use them only in limited circumstances such as operator overloading because in that case you need access to protected and private members, as discussed in the next section.

OPERATOR OVERLOADING

You often want to perform operations on objects, such as adding them, comparing them, or streaming them to or from files. For example, spreadsheets are really only useful when you can perform arithmetic actions on them, such as summing an entire row of cells.

Example: Implementing Addition for `SpreadsheetCells`

In true object-oriented fashion, `SpreadsheetCell` objects should be able to add themselves to other `SpreadsheetCell` objects. Adding a cell to another cell produces a third cell with the result. It doesn't change either of the original cells. The meaning of addition for `SpreadsheetCells` is the addition of the values of the cells. The string representations are ignored.

First Attempt: The `add` Method

You can declare and define an `add()` method for your `SpreadsheetCell` class like this:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    SpreadsheetCell add(const SpreadsheetCell& cell) const;
    // Omitted for brevity
};
```

This method adds two cells together, returning a new third cell whose value is the sum of the first two. It is declared `const` and takes a reference to a `const SpreadsheetCell` because `add()` does not change either of the source cells. Here is the implementation:

```
SpreadsheetCell SpreadsheetCell::add(const SpreadsheetCell& cell) const
{
    SpreadsheetCell newCell;
    newCell.set(mValue + cell.mValue); // update mValue and mString
    return newCell;
}
```

Note that the implementation creates a new `SpreadsheetCell` called `newCell` and returns a *copy* of that cell. You might be tempted to return a reference to the cell instead. However, that will not work because as soon as the `add()` method ends and `newCell` goes out of scope it will be destroyed. The reference that you returned will then be a dangling reference.

You can use the `add()` method like this:

```
SpreadsheetCell myCell(4), anotherCell(5);
SpreadsheetCell aThirdCell = myCell.add(anotherCell);
```

That works, but it's a bit clumsy. You can do better.

Second Attempt: Overloaded operator+ as a Method

It would be convenient to be able to add two cells with the plus sign the way that you add two `ints` or two `doubles`. Something like this:

```
SpreadsheetCell myCell(4), anotherCell(5);
SpreadsheetCell aThirdCell = myCell + anotherCell;
```

C++ allows you to write your own version of the plus sign, called the addition operator, to work correctly with your classes. To do that you write a method with the name `operator+` that looks like this:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    SpreadsheetCell operator+(const SpreadsheetCell& cell) const;
    // Omitted for brevity
};
```

NOTE You are allowed to write spaces between `operator` and the plus sign. For example, instead of writing `operator+`, you can write `operator +`. This is true for all operators. This book adopts the style without spaces.

The definition of the method is identical to the implementation of the `add()` method:

```
SpreadsheetCell
SpreadsheetCell::operator+(const SpreadsheetCell& cell) const
{
    SpreadsheetCell newCell;
    newCell.set(mValue + cell.mValue); // update mValue and mString.
    return newCell;
}
```

Now you can add two cells together using the plus sign as shown previously.

This syntax takes a bit of getting used to. Try not to worry too much about the strange method name `operator+` — it's just a name like `foo` or `add`. In order to understand the rest of the syntax, it helps to understand what's really going on. When your C++ compiler parses a program and encounters an operator, such as `+`, `-`, `=`, or `<<`, it tries to find a function or method with the name `operator+`, `operator-`, `operator=`, or `operator<<`, respectively, that takes the appropriate

parameters. For example, when the compiler sees the following line, it tries to find either a method in the `SpreadsheetCell` class named `operator+` that takes another `SpreadsheetCell` object or a global function named `operator+` that takes two `SpreadsheetCell` objects:

```
SpreadsheetCell aThirdCell = myCell + anotherCell;
```

If the `SpreadsheetCell` class contains an `operator+` method, then the previous line will be translated to:

```
SpreadsheetCell aThirdCell = myCell.operator+(anotherCell);
```

Note that there's no requirement that `operator+` takes as a parameter an object of the same type as the class for which it's written. You could write an `operator+` for `SpreadsheetCells` that takes a `Spreadsheet` to add to the `SpreadsheetCell`. That wouldn't make sense to the programmer, but the compiler would allow it.

Note also that you can give `operator+` any return value you want. Operator overloading is a form of function overloading, and recall that function overloading does not look at the return type of the function.

Implicit Conversions

Surprisingly, once you've written the `operator+` shown earlier, not only can you add two cells together, you can also add a cell to a `string`, a `double`, or an `int`!

```
SpreadsheetCell myCell(4), aThirdCell;
string str = "hello";
aThirdCell = myCell + str;
aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

The reason this code works is that the compiler does more to try to find an appropriate `operator+` than just look for one with the exact types specified. The compiler also tries to find an appropriate conversion for the types so that an `operator+` can be found. Constructors that take the type in question are appropriate converters. In the preceding example, when the compiler sees a `SpreadsheetCell` trying to add itself to `double`, it finds the `SpreadsheetCell` constructor that takes a `double` and constructs a temporary `SpreadsheetCell` object to pass to `operator+`. Similarly, when the compiler sees the line trying to add a `SpreadsheetCell` to a `string`, it calls the `string` `SpreadsheetCell` constructor to create a temporary `SpreadsheetCell` to pass to `operator+`.

This implicit conversion behavior is usually convenient. However, in the preceding example, it doesn't really make sense to add a `SpreadsheetCell` to a `string`. You can prevent the implicit construction of a `SpreadsheetCell` from a `string` by marking that constructor with the `explicit` keyword:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    explicit SpreadsheetCell(const std::string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
```

```
SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
// Remainder omitted for brevity
};
```

The `explicit` keyword goes only in the `class` definition, and only makes sense when applied to constructors that can be called with one argument, such as one-parameter constructors or multi-parameter constructors with default values for parameters.

The selection of an implicit constructor might be inefficient, because temporary objects must be created. To avoid implicit construction for adding a `double`, you could write a second `operator+` as follows:

```
SpreadsheetCell SpreadsheetCell::operator+(double rhs) const
{
    return SpreadsheetCell(mValue + rhs);
}
```

Note also that this demonstrates that you don't need to create a variable to return a value.

Third Attempt: Global operator+

Implicit conversions allow you to use an `operator+` method to add your `SpreadsheetCell` objects to `ints` and `doubles`. However, the operator is not commutative, as shown in the following code:

```
aThirdCell = myCell + 4;    // Works fine.
aThirdCell = myCell + 5.6; // Works fine.
aThirdCell = 4 + myCell;   // FAILS TO COMPILE!
aThirdCell = 5.6 + myCell; // FAILS TO COMPILE!
```

The implicit conversion works fine when the `SpreadsheetCell` object is on the left of the operator, but doesn't work when it's on the right. Addition is supposed to be commutative, so something is wrong here. The problem is that the `operator+` method must be called on a `SpreadsheetCell` object, and that object must be on the left-hand side of the `operator+`. That's just the way the C++ language is defined. So, there's no way you can get the above code to work with an `operator+` method.

However, you can get it to work if you replace the in-class `operator+` with a global `operator+` function that is not tied to any particular object. The function looks like this:

```
SpreadsheetCell operator+(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue + rhs.mValue); // update mValue and mString.
    return newCell;
}
```

Now all four of the addition lines work as you expect:

```
aThirdCell = myCell + 4;    // Works fine.
aThirdCell = myCell + 5.6; // Works fine.
aThirdCell = 4 + myCell;   // Works fine.
aThirdCell = 5.6 + myCell; // Works fine.
```

Note that the implementation of the global `operator+` accesses private data members of `SpreadsheetCell` objects. Therefore, it must be a friend function of the `SpreadsheetCell` class:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    //Omitted for brevity
};
```

You might be wondering what happens if you write the following code:

```
aThirdCell = 4.5 + 5.5;
```

It compiles and runs, but it's not calling the `operator+` you wrote. It does normal `double` addition of 4.5 and 5.5, which results in the following intermediate statement:

```
aThirdCell = 10;
```

To make this assignment work, there should be a `SpreadsheetCell` object on the right-hand side. The compiler will discover a user-defined constructor that takes a `double`, will use this constructor to implicitly convert the `double` value into a temporary `SpreadsheetCell` object, and will then call the assignment operator.

Overloading Arithmetic Operators

Now that you understand how to write `operator+`, the rest of the basic arithmetic operators are straightforward. Here are declarations of `-`, `*`, and `/` (you can also overload `%`, but it doesn't make sense for the `double` values stored in `SpreadsheetCells`):

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend SpreadsheetCell operator-(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend SpreadsheetCell operator*(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend SpreadsheetCell operator/(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    // Omitted for brevity
};
```

Here are the implementations. The only tricky aspect is remembering to check for division by zero. This implementation throws an exception if division by zero is detected:

```
SpreadsheetCell operator-(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue - rhs.mValue); // update mValue and mString.
    return newCell;
}
SpreadsheetCell operator*(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue * rhs.mValue); // update mValue and mString.
    return newCell;
}
SpreadsheetCell operator/(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    if (rhs.mValue == 0)
        throw invalid_argument("Divide by zero.");
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue / rhs.mValue); // update mValue and mString
    return newCell;
}
```

C++ does not require you to actually implement multiplication in `operator*`, division in `operator/`, and so on. You could implement multiplication in `operator/`, division in `operator+`, and so forth. However, that would be extremely confusing, and there is no good reason to do so except as a practical joke. Whenever possible, stick to the commonly used operator meanings in your implementations.

NOTE *In C++, you cannot change the precedence of operators. For example, * and / are always evaluated before + and -. The only thing user-defined operators can do is specify the implementation once the precedence of operations has been determined. C++ also does not allow you to invent new operator symbols.*

Overloading the Arithmetic Shorthand Operators

In addition to the basic arithmetic operators, C++ provides shorthand operators such as `+=` and `-=`. You might assume that writing `operator+` for your class provides `operator+=` also. No such luck. You have to overload the shorthand arithmetic operators explicitly. These operators differ from the basic arithmetic operators in that they change the object on the left-hand side of the operator instead of creating a new object. A second, subtler, difference is that, like the assignment operator, they generate a result that is a reference to the modified object.

The arithmetic shorthand operators always require an object on the left-hand side, so you should write them as methods, not as global functions. Here are the declarations for the `SpreadsheetCell` class:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    SpreadsheetCell& operator+=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator-=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator*=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator/=(const SpreadsheetCell& rhs);
    // Omitted for brevity
};
```

Here are the implementations:

```
SpreadsheetCell& SpreadsheetCell::operator+=(const SpreadsheetCell& rhs)
{
    set(mValue + rhs.mValue); // Call set to update mValue and mString.
    return *this;
}
SpreadsheetCell& SpreadsheetCell::operator-=(const SpreadsheetCell& rhs)
{
    set(mValue - rhs.mValue); // Call set to update mValue and mString.
    return *this;
}
SpreadsheetCell& SpreadsheetCell::operator*=(const SpreadsheetCell& rhs)
{
    set(mValue * rhs.mValue); // Call set to update mValue and mString.
    return *this;
}
SpreadsheetCell& SpreadsheetCell::operator/=(const SpreadsheetCell& rhs)
{
    if (rhs.mValue == 0)
        throw invalid_argument("Divide by zero.");
    set(mValue / rhs.mValue); // Call set to update mValue and mString.
    return *this;
}
```

The shorthand arithmetic operators are combinations of the basic arithmetic and the assignment operators. With the above definitions, you can now write code like this:

```
SpreadsheetCell myCell(4), aThirdCell(2);
aThirdCell -= myCell;
aThirdCell += 5.4;
```

You cannot, however, write code like this (which is a good thing!):

```
5.4 += aThirdCell;
```

Overloading Comparison Operators

The comparison operators, such as `>`, `<`, and `==`, are another useful set of operators to define for your classes. Like the basic arithmetic operators, they should be global friend functions so that you can use implicit conversion on both the left-hand side and right-hand side of the operator. The comparison operators all return a `bool`. Of course, you can change the return type, but that's not recommended. Here are the declarations and definitions:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend bool operator==(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    friend bool operator<(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    friend bool operator>(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    friend bool operator!=(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    friend bool operator<=(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    friend bool operator>=(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
    // Omitted for brevity
};

bool operator==(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue == rhs.mValue);
}
bool operator<(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue < rhs.mValue);
}
bool operator>(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue > rhs.mValue);
}
bool operator!=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue != rhs.mValue);
}
bool operator<=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue <= rhs.mValue);
}
bool operator>=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue >= rhs.mValue);
}
```

NOTE *The preceding overloaded operators are working with `mValue`, which is a double. Most of the time, performing equality or inequality tests on floating point values is not a good idea. You should use an epsilon test, but this falls outside the scope of this book.*

In classes with more data members, it might be painful to compare each data member. However, once you've implemented `==` and `<`, you can write the rest of the comparison operators in terms of those two. For example, here is a definition of `operator>=` that uses `operator<=`:

```
bool operator>=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return !(lhs < rhs);
}
```

The section on relational operators in Chapter 14 discusses a mechanism that automatically generates the operators `!=`, `>`, `<=`, and `>=` based on `operator ==` and `<`.

You can use these operators to compare `SpreadsheetCells` to other `SpreadsheetCells`, and to `doubles` and `ints`:

```
if (myCell > aThirdCell || myCell < 10) {
    cout << myCell.getValue() << endl;
}
```

Building Types with Operator Overloading

Many people find the syntax of operator overloading tricky and confusing, at least at first. The irony is that it's supposed to make things simpler. As you've discovered, that doesn't mean simpler for the person writing the class, but simpler for the person using the class. The point is to make your new classes as similar as possible to built-in types such as `int` and `double`: It's easier to add objects using `+` than to remember whether the method name you should call is `add()` or `sum()`.

NOTE *Provide operator overloading as a service to clients of your class.*

At this point, you might be wondering exactly which operators you can overload. The answer is "almost all of them — even some you've never heard of." You have actually just scratched the surface: You've seen the assignment operator in the section on object life cycles, the basic arithmetic operators, the shorthand arithmetic operators, and the comparison operators. Overloading the stream insertion and extraction operators is also useful. In addition, there are some tricky, but interesting, things you can do with operator overloading that you might not anticipate at first. The STL uses operator overloading extensively. Chapter 14 explains how and when to overload the rest of the operators. Chapter 15 and later chapters cover the STL.

BUILDING STABLE INTERFACES

Now that you understand all the gory syntax of writing classes in C++, it helps to revisit the design principles from Chapters 5 and 6. Classes are the main unit of abstraction in C++. You should apply the principles of abstraction to your classes to separate the interface from the implementation as much as possible. Specifically, you should make all data members private and provide getter and setter methods for them. This is how the `SpreadsheetCell` class is implemented. `mValue` and `mString` are private; `set()`, `getValue()`, and `getString()` set and retrieve those values. That way you can keep `mValue` and `mString` in sync internally without worrying about clients delving in and changing those values.

Using Interface and Implementation Classes

Even with the preceding measures and the best design principles, the C++ language is fundamentally unfriendly to the principle of abstraction. The syntax requires you to combine your `public` interfaces and `private` (or `protected`) data members and methods together in one class definition, thereby exposing some of the internal implementation details of the class to its clients. The downside of this is that if you have to add new `non-public` methods or data members to your class, all the clients of the class have to be recompiled. This can become a burden in bigger projects.

The good news is that you can make your interfaces a lot cleaner and hide all implementation details, resulting in stable interfaces. The bad news is that it takes a bit of hacking. The basic principle is to define two classes for every class you want to write: the *interface class* and the *implementation class*. The implementation class is identical to the class you would have written if you were not taking this approach. The interface class presents `public` methods identical to those of the implementation class, but it only has one data member: a pointer to an implementation class object. This is called the *pimpl idiom*, or private implementation idiom. The interface class method implementations simply call the equivalent methods on the implementation class object. The result of this is that no matter how the implementation changes, it has no impact on the `public` interface class. This reduces the need for recompilation. None of the clients that use the interface class need to be recompiled if the implementation (and only the implementation) changes. Note that this idiom only works if the single data member is a pointer to the implementation class. If it would be a `by-value` data member, the clients would have to be recompiled when the definition of the implementation class changes.

To use this approach with the `Spreadsheet` class, simply rename the old `Spreadsheet` class to `SpreadsheetImpl`. Here is the new `SpreadsheetImpl` class (which is identical to the old `Spreadsheet` class, but with a different name):

```
#include "SpreadsheetCell.h"
class SpreadsheetApplication; // Forward declaration
class SpreadsheetImpl
{
public:
    SpreadsheetImpl(const SpreadsheetApplication& theApp,
                    int inWidth = kMaxWidth, int inHeight = kMaxHeight);
    SpreadsheetImpl(const SpreadsheetImpl& src);
    ~SpreadsheetImpl();
    SpreadsheetImpl &operator=(const SpreadsheetImpl& rhs);
    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
```

```

        SpreadsheetCell& getCellAt(int x, int y);
        int getId() const;
        static const int kMaxHeight = 100;
        static const int kMaxWidth = 100;
    private:
        bool inRange(int val, int upper);
        void copyFrom(const SpreadsheetImpl& src);
        int mWidth, mHeight;
        int mId;
        SpreadsheetCell** mCells;
        const SpreadsheetApplication& mTheApp;
        static int sCounter;
    };
}

```

Then define a new `Spreadsheet` class that looks like this:

```

#include "SpreadsheetCell.h"
#include <memory>
// Forward declarations
class SpreadsheetImpl;
class SpreadsheetApplication;
class Spreadsheet
{
public:
    Spreadsheet(const SpreadsheetApplication& theApp, int inWidth,
               int inHeight);
    Spreadsheet(const SpreadsheetApplication& theApp);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();
    Spreadsheet& operator=(const Spreadsheet& rhs);
    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
    SpreadsheetCell& getCellAt(int x, int y);
    int getId() const;
private:
    std::unique_ptr<SpreadsheetImpl> mImpl;
};

```

This class now contains only one data member: a pointer to a `SpreadsheetImpl`. The public methods are identical to the old `Spreadsheet` with one exception: The `Spreadsheet` constructor with default arguments has been split into two constructors because the values for the default arguments were `const` members that are no longer in the `Spreadsheet` class. Instead, the `SpreadsheetImpl` class will provide the defaults.

The implementations of the `Spreadsheet` methods, such as `setCellAt()` and `getCellAt()`, just pass the request on to the underlying `SpreadsheetImpl` object:

```

void Spreadsheet::setCellAt(int x, int y, const SpreadsheetCell& inCell)
{
    mImpl->setCellAt(x, y, inCell);
}
SpreadsheetCell& Spreadsheet::getCellAt(int x, int y)
{
    return mImpl->getCellAt(x, y);
}

```

```
    }
    int Spreadsheet::getId() const
    {
        return mImpl->getId();
    }
```

The constructors for the `Spreadsheet` must construct a new `SpreadsheetImpl` to do its work. Note that the `SpreadsheetImpl` class has only one constructor with default arguments. Both normal constructors in the `Spreadsheet` class call that constructor on the `SpreadsheetImpl` class:

```
Spreadsheet::Spreadsheet(const SpreadsheetApplication& theApp, int inWidth,
    int inHeight)
{
    mImpl = std::make_unique<SpreadsheetImpl>(theApp, inWidth, inHeight);
}
Spreadsheet::Spreadsheet(const SpreadsheetApplication& theApp)
{
    mImpl = std::make_unique<SpreadsheetImpl>(theApp);
}
Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    mImpl = std::make_unique<SpreadsheetImpl>(*src.mImpl);
}
Spreadsheet::~Spreadsheet()
{}
```

The copy constructor looks a bit strange because it needs to copy the underlying `SpreadsheetImpl` from the source spreadsheet. Because the copy constructor takes a reference to a `SpreadsheetImpl`, not a pointer, you must dereference the `mImpl` pointer to get to the object itself so the constructor call can take its reference.

The `Spreadsheet` assignment operator must similarly pass on the assignment to the underlying `SpreadsheetImpl`:

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    *mImpl = *rhs.mImpl;
    return *this;
}
```

The first line in the assignment operator looks a little strange. You might be tempted to write this line instead:

```
mImpl = rhs.mImpl; // Incorrect assignment!
```

That code will not compile. The type of the `mImpl` data member in this implementation is `std::unique_ptr` which doesn't allow assignment. The `Spreadsheet` assignment operator needs to forward the call to the `SpreadsheetImpl` assignment operator, which only runs when you copy direct objects. By dereferencing the `mImpl` pointers, you force direct object assignment, which causes the assignment operator of `SpreadsheetImpl` to be called.

This technique to truly separate interface from implementation is powerful. Although a bit clumsy at first, once you get used to it you will find it natural to work with. However, it's not common practice in most workplace environments, so you might find some resistance to trying it from your coworkers. The most compelling argument in favor of it is not the aesthetic one of splitting out the interface but the cost of a full rebuild if the implementation of the class changes. A full rebuild on a huge project might take hours. With stable interface classes, rebuild time is reduced.

SUMMARY

This chapter, along with Chapter 7, provided all the tools you need to write solid, well-designed classes, and to use objects effectively.

You discovered that dynamic memory allocation in objects presents new challenges: You must free the memory in the destructor, copy the memory in the copy constructor, and both free and copy the memory in the assignment operator. You learned how to prevent assignment and pass-by-value by explicitly deleting the copy constructor and assignment operator.

You learned more about different kinds of data members, including `static`, `const`, `const reference`, and `mutable` members. You also learned about `static`, `inline`, and `const` methods, method overloading and default parameters. The chapter also described nested class definitions, and `friend` classes, functions and methods.

You encountered operator overloading, and learned how to overload the arithmetic and comparison operators, both as global `friend` functions and as class methods.

Finally, you learned how to take abstraction to an extreme by providing separate interface and implementation classes.

Now that you're fluent in the language of object-oriented programming, it's time to tackle inheritance, which is covered in Chapter 9.

9

Discovering Inheritance Techniques

WHAT'S IN THIS CHAPTER?

- How to extend a class through inheritance
- How to employ inheritance to reuse code
- How to build interactions between base classes and derived classes
- How to use inheritance to achieve polymorphism
- How to work with multiple inheritance
- How to deal with unusual problems in inheritance

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Without inheritance, classes would simply be data structures with associated behaviors. That alone would be a powerful improvement over procedural languages, but inheritance adds an entirely new dimension. Through inheritance, you can build new classes based on existing ones. In this way, your classes become reusable and extensible components. This chapter will teach you the different ways to leverage the power of inheritance. You will learn about the specific syntax of inheritance as well as sophisticated techniques for making the most of inheritance.

The portion of this chapter relating to polymorphism draws heavily on the spreadsheet example discussed in Chapters 7 and 8. This chapter also refers to the object-oriented methodologies described in Chapter 5. If you have not read that chapter and are unfamiliar with the theories behind inheritance, you should review Chapter 5 before continuing.

BUILDING CLASSES WITH INHERITANCE

In Chapter 5, you learned that an “is-a” relationship recognizes the pattern that real-world objects tend to exist in hierarchies. In programming, that pattern becomes relevant when you need to write a class that builds on, or slightly changes, another class. One way to accomplish this aim is to copy code from one class and paste it into the other. By changing the relevant parts or amending the code, you can achieve the goal of creating a new class that is slightly different from the original. This approach, however, leaves an OOP programmer feeling sullen and slightly annoyed for the following reasons:

- A bug fix to the original class will not be reflected in the new class because the two classes contain completely separate code.
- The compiler does not know about any relationship between the two classes, so they are not polymorphic — they are not just different variations on the same thing.
- This approach does not build a true is-a relationship. The new class is very similar to the original because it shares code, not because it really *is* the same type of object.
- The original code might not be obtainable. It may exist only in a precompiled binary format, so copying and pasting the code might be impossible.

Not surprisingly, C++ provides built-in support for defining a true is-a relationship. The characteristics of C++ is-a relationships are described in the following section.

Extending Classes

When you write a class definition in C++, you can tell the compiler that your class is *inheriting from*, or *extending*, an existing class. By doing so, your class will automatically contain the data members and methods of the original class, which is called the *parent class* or *base class* or *superclass*. Extending an existing class gives your class (which is now called a *derived class* or a *subclass*) the ability to describe only the ways in which it is different from the parent class.

To extend a class in C++, you specify the class you are extending when you write the class definition. To show the syntax for inheritance, I use two classes called `Super` and `Sub`. Don’t worry — more interesting examples are coming later. To begin, consider the following definition for the `Super` class:

```
class Super
{
    public:
        Super();
        void someMethod();
    protected:
        int mProtectedInt;
    private:
        int mPrivateInt;
};
```

If you wanted to build a new class, called `Sub`, which inherits from `Super`, you would tell the compiler that `Sub` derives from `Super` with the following syntax:

```
class Sub : public Super
{
public:
    Sub();
    void someOtherMethod();
};
```

`Sub` itself is a full-fledged class that just happens to share the characteristics of the `Super` class. Don't worry about the word `public` for now — its meaning is explained later in this chapter. Figure 9-1 shows the simple relationship between `Sub` and `Super`. You can declare objects of type `Sub` just like any other object. You could even define a third class that inherits from `Sub`, forming a chain of classes, as shown in Figure 9-2.

`Sub` doesn't have to be the only derived class of `Super`. Additional classes can also derive from `Super`, effectively becoming *siblings* to `Sub`, as shown in Figure 9-3.

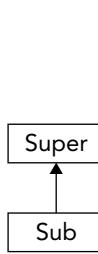


FIGURE 9-1

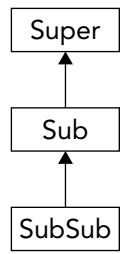


FIGURE 9-2

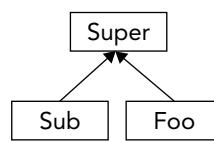


FIGURE 9-3

A Client's View of Inheritance

To a client, or another part of your code, an object of type `Sub` is also an object of type `Super` because `Sub` inherits from `Super`. This means that all the `public` methods and data members of `Super` *and* all the `public` methods and data members of `Sub` are available.

Code that uses the derived class does not need to know which class in your inheritance chain has defined a method in order to call it. For example, the following code calls two methods of a `Sub` object even though one of the methods was defined by the `Super` class:

```
Sub mySub;
mySub.someMethod();
mySub.someOtherMethod();
```

It is important to understand that inheritance works in only one direction. The `Sub` class has a very clearly defined relationship to the `Super` class, but the `Super` class, as written, doesn't know anything about the `Sub` class. That means that objects of type `Super` do not support `public` methods and data members of `Sub` because `Super` is *not* a `Sub`.

The following code will not compile because the `Super` class does not contain a `public` method called `someOtherMethod()`:

```
Super mySuper;
mySuper.someOtherMethod(); // Error! Super doesn't have a someOtherMethod().
```

NOTE *From the perspective of other code, an object belongs to its defined class as well as to any base classes.*

A pointer or reference to an object can refer to an object of the declared class or any of its derived classes. This tricky subject is explained in detail later in this chapter. The concept to understand at this point is that a pointer to a `Super` can actually be pointing to a `Sub` object. The same is true for a reference. The client can still access only the methods and data members that exist in `Super`, but through this mechanism, any code that operates on a `Super` can also operate on a `Sub`.

For example, the following code compiles and works just fine even though it initially appears that there is a type mismatch:

```
Super* superPointer = new Sub(); // Create sub, store it in super pointer.
```

However, you cannot call methods from the `Sub` class through the `Super` pointer. The following will not work:

```
superPointer->someOtherMethod();
```

This will be flagged as an error by the compiler, because, although the object is of type `Sub` and therefore does have `someOtherMethod()` defined, the compiler can only think of it as type `Super` which does not have `someOtherMethod()` defined.

A Derived Class's View of Inheritance

To the derived class itself, nothing much has changed in terms of how it is written or how it behaves. You can still define methods and data members on a derived class just as you would on a regular class. The previous definition of `Sub` declares a method called `someOtherMethod()`. Thus, the `Sub` class augments the `Super` class by adding an additional method.

A derived class can access `public` and `protected` methods and data members declared in its base class as though they were its own, because technically, they are. For example, the implementation of `someOtherMethod()` on `Sub` could make use of the data member `mProtectedInt`, which was declared as part of `Super`. The following code shows this implementation. Accessing a base class data member or method is no different than if the data member or method were declared as part of the derived class.

```
void Sub::someOtherMethod()
{
    cout << "I can access base class data member mProtectedInt." << endl;
    cout << "Its value is " << mProtectedInt << endl;
}
```

When access specifiers (`public`, `private`, and `protected`) were introduced in Chapter 7, the difference between `private` and `protected` may have been confusing. Now that you understand derived classes, the difference should be clear. If a class declares methods or data members as

protected, derived classes have access to them. If they are declared as `private`, derived classes do not have access.

The following implementation of `someOtherMethod()` will not compile because the derived class attempts to access a `private` data member from the base class.

```
void Sub::someOtherMethod()
{
    cout << "I can access base class data member mProtectedInt." << endl;
    cout << "Its value is " << mProtectedInt << endl;
    cout << "The value of mPrivateInt is " << mPrivateInt << endl; // Error!
}
```

The `private` access specifier gives you control over how a potential derived class could interact with your class. I recommend that you make all your data members `private` by default. You can provide `public` getters and setters if you want to allow anyone to access those data members, and you can provide `protected` getters and setters if you only want derived classes to access them. The reason to make data members `private` by default is that this provides the highest level of encapsulation. This means that you can change how you represent your data while keeping the `public` and `protected` interface unchanged. Without giving direct access to data members, you can also easily add checks on the input data in your `public` and `protected` setters. Methods should also be `private` by default. Only make those methods `public` that are designed to be `public`, and make methods `protected` if you only want derived classes to have access to them.

NOTE *From the perspective of a derived class, all public and protected data members and methods from the base class are available for use.*

Preventing Inheritance

C++ allows you to mark a class as `final`, which means trying to inherit from it will result in a compiler error. A class can be marked as `final` with the `final` keyword right behind the name of the class. For example, the following `Super` class is marked as `final`:

```
class Super final
{
    // Omitted for brevity
};
```

The following `Sub` class tries to inherit from the `Super` class, but this will result in a compiler error because `Super` is marked as `final`.

```
class Sub : public Super
{
    // Omitted for brevity
};
```

Overriding Methods

The main reasons to inherit from a class are to add or replace functionality. The definition of `Sub` adds functionality to its parent class by providing an additional method, `someOtherMethod()`. The other method, `someMethod()`, is inherited from `Super` and behaves in the derived class exactly as it does in the base class. In many cases, you will want to modify the behavior of a class by replacing, or *overriding*, a method.

How I Learned to Stop Worrying and Make Everything `virtual`

There is one small twist to overriding methods in C++ and it has to do with the keyword `virtual`. Only methods that are declared as `virtual` in the base class can be overridden properly by derived classes. The keyword goes at the beginning of a method declaration as shown in the modified version of `Super` that follows:

```
class Super
{
public:
    Super();
    virtual void someMethod();
protected:
    int mProtectedInt;
private:
    int mPrivateInt;
};
```

The `virtual` keyword has a few subtleties and is often cited as a poorly designed part of the language. A good rule of thumb is to just make all of your methods `virtual`. That way, you won't have to worry about whether or not overriding the method will work. The only drawback is a very tiny performance hit. The subtleties of the `virtual` keyword are covered toward the end of this chapter, and performance is discussed further in Chapter 25.

Even though it is unlikely that the `Sub` class will be extended, it is a good idea to make its methods `virtual` as well, just in case.

```
class Sub : public Super
{
public:
    Sub();
    virtual void someOtherMethod();
};
```

NOTE *As a rule of thumb, make all your methods `virtual` (including the destructor, but not constructors) to avoid problems associated with omission of the `virtual` keyword.*

Syntax for Overriding a Method

To override a method, you redeclare it in the derived class definition exactly as it was declared in the base class. In the derived class's implementation file, you provide the new definition.

For example, the `Super` class contains a method called `someMethod()`. The definition of `someMethod()` is provided in `Super.cpp` and shown here:

```
void Super::someMethod()
{
    cout << "This is Super's version of someMethod()." << endl;
}
```

Note that you do not repeat the `virtual` keyword in front of the method definition.

If you wish to provide a new definition for `someMethod()` in the `Sub` class, you must first add it to the class definition for `Sub`, as follows:

```
class Sub : public Super
{
public:
    Sub();
    virtual void someMethod() override; // Overrides Super's someMethod()
    virtual void someOtherMethod();
};
```

It is recommended to add the `override` keyword to the end of overriding method declarations. A more detailed discussion of the `override` keyword comes later in this chapter. The new definition of `someMethod()` is specified along with the rest of `Sub`'s methods in `Sub.cpp`.

```
void Sub::someMethod()
{
    cout << "This is Sub's version of someMethod()." << endl;
}
```

Once a method or destructor is marked as `virtual`, it will be `virtual` for all derived classes even if the `virtual` keyword is removed from derived classes. For example, in the following `Sub` class, `someMethod()` is still `virtual` and can still be overridden by derived classes of `Sub`, because it was marked as `virtual` in the `Super` class.

```
class Sub : public Super
{
public:
    Sub();
    void someMethod() override; // Overrides Super's someMethod()
};
```

A Client's View of Overridden Methods

With the preceding changes, other code would still call `someMethod()` the same way it did before. Just as before, the method could be called on an object of class `Super` or an object of class `Sub`. Now, however, the behavior of `someMethod()` will vary based on the class of the object.

For example, the following code works just as it did before, calling `Super`'s version of `someMethod()`:

```
Super mySuper;  
mySuper.someMethod(); // Calls Super's version of someMethod().
```

The output of this code is:

```
This is Super's version of someMethod().
```

If the code declares an object of class `Sub`, the other version will automatically be called:

```
Sub mySub;  
mySub.someMethod(); // Calls Sub's version of someMethod()
```

The output this time is:

```
This is Sub's version of someMethod().
```

Everything else about objects of class `Sub` remains the same. Other methods that might have been inherited from `Super` will still have the definition provided by `Super` unless they are explicitly overridden in `Sub`.

As you learned earlier, a pointer or reference can refer to an object of a class or any of its derived classes. The object itself “knows” the class of which it is actually a member, so the appropriate method is called as long as it was declared `virtual`. For example, if you have a `Super` reference that refers to an object that is really a `Sub`, calling `someMethod()` will actually call the derived class's version, as shown next. This aspect of overriding will *not* work properly if you omit the `virtual` keyword in the base class.

```
Sub mySub;  
Super& ref = mySub;  
ref.someMethod(); // Calls Sub's version of someMethod()
```

Remember that even though a base class reference or pointer knows that it is actually a derived class, you cannot access derived class methods or members that are not defined in the base class. The following code will not compile because a `Super` reference does not have a method called `someOtherMethod()`:

```
Sub mySub;  
Super& ref = mySub;  
mySub.someOtherMethod(); // This is fine.  
ref.someOtherMethod(); // Error
```

The derived class knowledge characteristic is *not* true for nonpointer nonreference objects. You can cast or assign a `Sub` to a `Super` because a `Sub` is a `Super`. However, the object will lose any knowledge of the derived class at this point:

```
Sub mySub;  
Super assignedObject = mySub; // Assigns a Sub to a Super.  
assignedObject.someMethod(); // Calls Super's version of someMethod()
```

One way to remember this seemingly strange behavior is to imagine what the objects look like in memory. Picture a `Super` object as a box taking up a certain amount of memory. A `Sub` object is a box that is a little bit bigger because it has everything a `Super` has plus a bit more. When you have a reference or pointer to a `Sub`, the box doesn't change — you just have a new way of accessing it. However, when you cast a `Sub` into a `Super`, you are throwing out all the “uniqueness” of the `Sub` class to fit it into a smaller box.

NOTE *Derived classes retain their overridden methods when referred to by base class pointers or references. They lose their uniqueness when cast to a base class object. The loss of overridden methods and derived class data is called slicing.*

Preventing Overriding

C++ allows you to mark a method as `final` which means that the method cannot be overridden in a derived class. Trying to override a `final` method will result in a compiler error. Take the following `Super` class:

```
class Super
{
public:
    Super();
    virtual void someMethod() final;
};
```

Trying to override `someMethod()`, as in the following `Sub` class, will result in a compiler error because `someMethod()` is marked as `final` in the `Super` class.

```
class Sub : public Super
{
public:
    Sub();
    virtual void someMethod() override; // Error
    virtual void someOtherMethod();
};
```

INHERITANCE FOR REUSE

Now that you are familiar with the basic syntax for inheritance, it's time to explore one of the main reasons that inheritance is an important feature of the C++ language. Inheritance is a vehicle that allows you to leverage existing code. This section presents an example of inheritance for the purpose of code reuse.

The WeatherPrediction Class

Imagine that you are given the task of writing a program to issue simple weather predictions, working with both Fahrenheit and Celsius. Weather predictions may be a little out of your area

of expertise as a programmer, so you obtain a third-party class library that was written to make weather predictions based on the current temperature and the present distance between Jupiter and Mars (hey, it's plausible). This third-party package is distributed as a compiled library to protect the intellectual property of the prediction algorithms, but you do get to see the class definition. The class definition for `WeatherPrediction` is as follows:

```
// Predicts the weather using proven new-age techniques given the current
// temperature and the distance from Jupiter to Mars. If these values are
// not provided, a guess is still given but it's only 99% accurate.
class WeatherPrediction
{
    public:
        // Virtual destructor
        virtual ~WeatherPrediction();
        // Sets the current temperature in fahrenheit
        virtual void setCurrentTempFahrenheit(int inTemp);
        // Sets the current distance between Jupiter and Mars
        virtual void setPositionOfJupiter(int inDistanceFromMars);
        // Gets the prediction for tomorrow's temperature
        virtual int getTomorrowTempFahrenheit() const;
        // Gets the probability of rain tomorrow. 1 means
        // definite rain. 0 means no chance of rain.
        virtual double getChanceOfRain() const;
        // Displays the result to the user in this format:
        // Result: x.xx chance. Temp. xx
        virtual void showResult() const;
        // Returns string representation of the temperature
        virtual std::string getTemperature() const;
    private:
        int mCurrentTempFahrenheit;
        int mDistanceFromMars;
};
```

Note that this class marks all methods as `virtual`, because the class presumes that its methods might be overridden in a derived class.

This class solves most of the problems for your program. However, as is usually the case, it's not *exactly* right for your needs. First, all the temperatures are given in Fahrenheit. Your program needs to operate in Celsius as well. Also, the `showResult()` method might not display the result in a way you require.

Adding Functionality in a Derived Class

When you learned about inheritance in Chapter 5, adding functionality was the first technique described. Fundamentally, your program needs something just like the `WeatherPrediction` class but with a few extra bells and whistles. Sounds like a good case for inheritance to reuse code. To begin, define a new class, `MyWeatherPrediction`, that inherits from `WeatherPrediction`.

```
#include "WeatherPrediction.h"
class MyWeatherPrediction : public WeatherPrediction
{
```

The preceding class definition will compile just fine. The `MyWeatherPrediction` class can already be used in place of `WeatherPrediction`. It will provide the same functionality, but nothing new yet. For the first modification, you might want to add knowledge of the Celsius scale to the class. There is a bit of a quandary here because you don't know what the class is doing internally. If all of the internal calculations are made by using Fahrenheit, how do you add support for Celsius? One way is to use the derived class to act as a go-between, interfacing between the user, who can use either scale, and the base class, which only understands Fahrenheit.

The first step in supporting Celsius is to add new methods that allow clients to set the current temperature in Celsius instead of Fahrenheit and to get tomorrow's prediction in Celsius instead of Fahrenheit. You will also need private helper methods that convert between Celsius and Fahrenheit. These methods can be static because they are the same for all instances of the class.

```
#include "WeatherPrediction.h"
class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int inTemp);
    virtual int getTomorrowTempCelsius() const;
private:
    static int convertCelsiusToFahrenheit(int inCelsius);
    static int convertFahrenheitToCelsius(int inFahrenheit);
};
```

The new methods follow the same naming convention as the parent class. Remember that from the point of view of other code, a `MyWeatherPrediction` object will have all of the functionality defined in both `MyWeatherPrediction` and `WeatherPrediction`. Adopting the parent class's naming convention presents a consistent interface.

The implementation of the Celsius/Fahrenheit conversion methods is left as an exercise for the reader — and a fun one at that! The other two methods are more interesting. To set the current temperature in Celsius, you need to convert the temperature first and then present it to the parent class in units that it understands.

```
void MyWeatherPrediction::setCurrentTempCelsius(int inTemp)
{
    int fahrenheitTemp = convertCelsiusToFahrenheit(inTemp);
    setCurrentTempFahrenheit(fahrenheitTemp);
}
```

As you can see, once the temperature is converted, the method calls the existing functionality from the base class. Similarly, the implementation of `getTomorrowTempCelsius()` uses the parent's existing functionality to get the temperature in Fahrenheit, but converts the result before returning it.

```
int MyWeatherPrediction::getTomorrowTempCelsius() const
{
    int fahrenheitTemp = getTomorrowTempFahrenheit();
    return convertFahrenheitToCelsius(fahrenheitTemp);
}
```

The two new methods effectively reuse the parent class because they “wrap” the existing functionality in a way that provides a new interface for using it.

You can also add new functionality completely unrelated to existing functionality of the parent class. For example, you could add a method that will retrieve alternative forecasts from the Internet or a method that will suggest an activity based on the predicted weather.

Replacing Functionality in a Derived Class

The other major technique for inheritance is replacing existing functionality. The `showResult()` method in the `WeatherPrediction` class is in dire need of a facelift. `MyWeatherPrediction` can override this method to replace the behavior with its own implementation.

The new class definition for `MyWeatherPrediction` is as follows:

```
#include "WeatherPrediction.h"
class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int inTemp);
    virtual int getTomorrowTempCelsius() const;
    virtual void showResult() const override;
private:
    static int convertCelsiusToFahrenheit(int inCelsius);
    static int convertFahrenheitToCelsius(int inFahrenheit);
};
```

A possible new user-friendly implementation follows:

```
void MyWeatherPrediction::showResult() const
{
    cout << "Tomorrow's temperature will be " <<
        getTomorrowTempCelsius() << " degrees Celsius (" <<
        getTomorrowTempFahrenheit() << " degrees Fahrenheit)" << endl;
    cout << "Chance of rain is " << (getChanceOfRain() * 100) << " percent"
        << endl;
    if (getChanceOfRain() > 0.5) {
        cout << "Bring an umbrella!" << endl;
    }
}
```

To clients using this class, it's as if the old version of `showResult()` never existed. As long as the object is a `MyWeatherPrediction` object, the new version will be called.

As a result of these changes, `MyWeatherPrediction` has emerged as a new class with new functionality tailored to a more specific purpose. Yet, it did not require much code because it leveraged its base class's existing functionality.

RESPECT YOUR PARENTS

When you write a derived class, you need to be aware of the interaction between parent classes and child classes. Issues such as order of creation, constructor chaining, and casting are all potential sources of bugs.

Parent Constructors

Objects don't spring to life all at once; they must be constructed along with their parents and any objects that are contained within them. C++ defines the creation order as follows:

1. If the class has a base class, the default constructor of the base class is executed, unless there is a call to a base class constructor in the ctor-initializer in which case that constructor is called instead of the default constructor.
2. Non-static data members of the class are constructed in the order in which they are declared.
3. The body of the class's constructor is executed.

These rules can apply recursively. If the class has a grandparent, the grandparent is initialized before the parent, and so on. The following code shows this creation order. As a reminder, I generally advise against implementing methods directly in a class definition, as I've done in the code that follows. In the interest of readable and concise examples, I have broken my own rule. The proper execution will output the result 123.

```
class Something
{
public:
    Something() { cout << "2"; }
};

class Parent
{
public:
    Parent() { cout << "1"; }
};

class Child : public Parent
{
public:
    Child() { cout << "3"; }
private:
    Something mDataMember;
};

int main()
{
    Child myChild;
    return 0;
}
```

When the `myChild` object is created, the constructor for `Parent` is called first, outputting the string "1". Next, `mDataMember` is initialized, calling the `Something` constructor, which outputs the string "2". Finally, the `Child` constructor is called, which outputs "3".

Note that the `Parent` constructor was called automatically. C++ will automatically call the default constructor for the parent class if one exists. If no default constructor exists in the parent class, or if one does exist but you wish to use an alternate constructor, you can *chain* the constructor just as when initializing data members in the constructor initializer.

The following code shows a version of `Super` that lacks a default constructor. The associated version of `Sub` must explicitly tell the compiler how to call the `Super` constructor or the code will not compile.

```
class Super
{
    public:
        Super(int i);
};

class Sub : public Super
{
    public:
        Sub();
};

Sub::Sub() : Super(7)
{
    // Do Sub's other initialization here.
}
```

In the preceding code, the `Sub` constructor passes a fixed value (7) to the `Super` constructor. `Sub` could also pass a variable if its constructor required an argument:

```
Sub::Sub(int i) : Super(i) {}
```

Passing constructor arguments from the derived class to the base class is perfectly fine and quite normal. Passing data members, however, will not work. The code will compile, but remember that data members are not initialized until *after* the base class is constructed. If you pass a data member as an argument to the parent constructor, it will be uninitialized.

WARNING *Virtual methods behave differently in constructors. If your derived class has overridden a virtual method from the base class, calling that method from a base class constructor will call the base class implementation of that virtual method and not your overridden version in the derived class.*

Parent Destructors

Because destructors cannot take arguments, the language can always automatically call the destructor for parent classes. The order of destruction is conveniently the reverse of the order of construction:

1. The body of the class's destructor is called.
2. Any data members of the class are destroyed in the reverse order of their construction.
3. The parent class, if any, is destructed.

Again, these rules apply recursively. The lowest member of the chain is always destructed first. The following code adds destructors to the previous example. The destructors are all declared `virtual`, which is very important and will be discussed right after this example. If executed, this code will output "123321".

```
class Something
{
public:
    Something() { cout << "2"; }
    virtual ~Something() { cout << "2"; }
};

class Parent
{
public:
    Parent() { cout << "1"; }
    virtual ~Parent() { cout << "1"; }
};

class Child : public Parent
{
public:
    Child() { cout << "3"; }
    virtual ~Child() { cout << "3"; }
private:
    Something mDataMember;
};

int main()
{
    Child myChild;
    return 0;
}
```

Notice that the destructors are all `virtual`. As a rule of thumb, all destructors should be declared `virtual`. If the preceding destructors were not declared `virtual`, the code would continue to work fine. However, if code ever called `delete` on a base class pointer that was really pointing to a derived class, the destruction chain would begin in the wrong place. For example, the following code is similar to the previous example but the destructors are not `virtual`. This becomes a problem when a `Child` object is accessed as a pointer to a `Parent` and deleted.

```
class Something
{
public:
    Something() { cout << "2"; }
    ~Something() { cout << "2"; } // Should be virtual, but will work
};

class Parent
{
public:
    Parent() { cout << "1"; }
    ~Parent() { cout << "1"; } // BUG! Make this virtual!
};

class Child : public Parent
{
```

```

public:
    Child() { cout << "3"; }
    ~Child() { cout << "3"; } // Should be virtual, but will work
private:
    Something mDataMember;
};

int main()
{
    Parent* ptr = new Child();
    delete ptr;
    return 0;
}

```

The output of this code is a shockingly terse "1231". When the `ptr` variable is deleted, only the `Parent` destructor is called because the destructor was not declared `virtual`. As a result, the `Child` destructor is not called and the destructors for its data members are not called.

Technically, you could fix the preceding problem by making the `Parent` destructor `virtual`. The "virtualness" would automatically be used by any children. However, I advocate explicitly making all destructors `virtual` so that you never have to worry about it.

WARNING *Always make your destructors `virtual`! The compiler generated default destructor is not `virtual`, so you should define your own `virtual` destructor, at least for your parent classes.*

WARNING *Just as with constructors, virtual methods behave differently when called from a destructor. If your derived class has overridden a virtual method from the base class, calling that method from the base class destructor will call the base class implementation of that virtual method and not your overridden version in the derived class.*

Referring to Parent Names

When you override a method in a derived class, you are effectively replacing the original as far as other code is concerned. However, that parent version of the method still exists and you may want to make use of it. For example, an overridden method would like to keep doing what the base class implementation does, plus something else. Take a look at the `getTemperature()` method in the `WeatherPrediction` class that returns a `string` representation of the current temperature.

```

class WeatherPrediction
{
public:
    virtual std::string getTemperature() const;
    // Omitted for brevity
};

```

You can override this method in the `MyWeatherPrediction` class as follows:

```
class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual std::string getTemperature() const override;
    // Omitted for brevity
};
```

Suppose the derived class wants to add $^{\circ}$ F to the string by first calling the base class `getTemperature()` method and then adding $^{\circ}$ F to the string. You might want to try to write this as follows:

```
string MyWeatherPrediction::getTemperature() const
{
    // Note: \u00B0 is the ISO/IEC 10646 representation of the degree symbol.
    return getTemperature() + "\u00B0F"; // BUG
}
```

However, this will not work because, under the rules of name resolution for C++, it first resolves against the local scope, then the class scope, and as a consequence will end up calling `MyWeatherPrediction::getTemperature()`. This will result in an infinite recursion until you run out of stack space (some compilers detect this error and report it at compile time).

To make this work, you need to use the scope resolution operator as follows:

```
string MyWeatherPrediction::getTemperature() const
{
    // Note: \u00B0 is the ISO/IEC 10646 representation of the degree symbol.
    return WeatherPrediction::getTemperature() + "\u00B0F";
}
```

NOTE Microsoft Visual C++ supports the `_super` keyword (with two underscores). This allows you to write the following:

```
return __super::getTemperature() + "\u00B0F";
```

Calling the parent version of the current method is a commonly used pattern in C++. If you have a chain of derived classes, each might want to perform the operation already defined by the base class but add their own additional functionality as well.

As another example, imagine a class hierarchy of types of books. A diagram showing such a hierarchy is shown in Figure 9-4.

Since each lower class in the hierarchy further specifies the type of book, a method that gets the description of a book really needs to take all levels of the hierarchy into consideration. This can be accomplished by

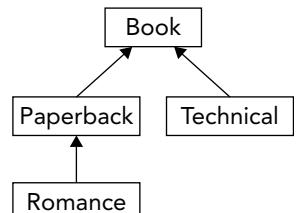


FIGURE 9-4

chaining to the parent method. The following code illustrates this pattern. The code also defines a `virtual getHeight()` method, discussed further after the example.

```

class Book
{
public:
    virtual ~Book() {}
    virtual string getDescription() const { return "Book"; }
    virtual int getHeight() const { return 120; }
};

class Paperback : public Book
{
public:
    virtual string getDescription() const override {
        return "Paperback " + Book::getDescription();
    }
};

class Romance : public Paperback
{
public:
    virtual string getDescription() const override {
        return "Romance " + Paperback::getDescription();
    }
    virtual int getHeight() const override {
        return Paperback::getHeight() / 2; }
};

class Technical : public Book
{
public:
    virtual string getDescription() const override {
        return "Technical " + Book::getDescription();
    }
};

int main()
{
    Romance novel;
    Book book;
    cout << novel.getDescription() << endl; // Outputs "Romance Paperback Book"
    cout << book.getDescription() << endl; // Outputs "Book"
    cout << novel.getHeight() << endl; // Outputs "60"
    cout << book.getHeight() << endl; // Outputs "120"
    return 0;
}

```

The `Book` class defines a `virtual getHeight()` method, returning 120. Only the `Romance` class overrides this by calling `getHeight()` on its parent class (`Paperback`) and dividing the result by two as follows:

```
virtual int getHeight() const override { return Paperback::getHeight() / 2; }
```

However, `Paperback` does not override `getHeight()`, so C++ will walk up the class hierarchy to find a class that implements `getHeight()`. In the preceding example, `Paperback::getHeight()` will resolve to `Book::getHeight()`.

Casting Up and Down

As you have already seen, an object can be cast or assigned to its parent class. If the cast or assignment is performed on a plain old object, this results in slicing:

```
Super mySuper = mySub; // SLICE!
```

Slicing occurs in situations like this because the end result is a `Super` object, and `Super` objects lack the additional functionality defined in the `Sub` class. However, slicing does *not* occur if a derived class is assigned to a pointer or reference to its base class:

```
Super& mySuper = mySub; // No slice!
```

This is generally the correct way to refer to a derived class in terms of its base class, also called *upcasting*. This is why it's always a good idea to make your methods and functions take references to classes instead of directly using objects of those classes. By using references, derived classes can be passed in without slicing.

WARNING *When upcasting, use a pointer or reference to the base class to avoid slicing.*

Casting from a base class to one of its derived classes, also called *downcasting*, is often frowned upon by professional C++ programmers because there is no guarantee that the object really belongs to that derived class, and because downcasting is a sign of bad design. For example, consider the following code:

```
void presumptuous(Super* inSuper)
{
    Sub* mySub = static_cast<Sub*>(inSuper);
    // Proceed to access Sub methods on mySub.
}
```

If the author of `presumptuous()` also wrote code that called `presumptuous()`, everything would probably be okay because the author knows that the function expects the argument to be of type `Sub*`. However, if other programmers were to call `presumptuous()`, they might pass in a `Super*`. There are no compile-time checks that can be done to enforce the type of the argument, and the function blindly assumes that `inSuper` is actually a pointer to a `Sub`.

Downcasting is sometimes necessary, and you can use it effectively in controlled circumstances. If you're going to downcast, however, you should use a `dynamic_cast`, which uses the object's built-in knowledge of its type to refuse a cast that doesn't make sense. If a `dynamic_cast` fails on a pointer, the pointer's value will be `nullptr` instead of pointing to nonsensical data. If a `dynamic_cast` fails on an object reference, a `std::bad_cast` exception will be thrown. Chapter 10 discusses casting in more detail.

The previous example should have been written as follows:

```
void lessPresumptuous(Super* inSuper)
{
    Sub* mySub = dynamic_cast<Sub*>(inSuper);
    if (mySub != nullptr) {
        // Proceed to access Sub methods on mySub.
    }
}
```

WARNING Use *downcasting* only when necessary and be sure to use a `dynamic_cast`.

INHERITANCE FOR POLYMORPHISM

Now that you understand the relationship between a derived class and its parent, you can use inheritance in its most powerful scenario — polymorphism. Chapter 5 discusses how polymorphism allows you to use objects with a common parent class interchangeably, and to use objects in place of their parents.

Return of the Spreadsheet

Chapters 7 and 8 use a spreadsheet program as an example of an application that lends itself to an object-oriented design. A `SpreadsheetCell` represents a single element of data. That element could be either a `double` or a `string`. A simplified class definition for `SpreadsheetCell` follows. Note that a cell can be set either as a `double` or a `string`. The current value of the cell, however, is always returned as a `string` for this example.

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    virtual void set(double inDouble);
    virtual void set(const std::string& inString);
    virtual std::string getString() const;
private:
    static std::string doubleToString(double inValue);
    static double stringtoDouble(const std::string& inString);
    double      mValue;
    std::string mString;
};
```

The preceding `SpreadsheetCell` class seems to be having an identity crisis — sometimes a cell represents a `double`, sometimes a `string`. Sometimes it has to convert between these formats. To achieve this duality, the class needs to store both values even though a given cell should be able to contain only a single value. Worse still, what if additional types of cells are needed, such as a

formula cell or a date cell? The `SpreadsheetCell` class would grow dramatically to support all of these data types and the conversions between them.

Designing the Polymorphic Spreadsheet Cell

The `SpreadsheetCell` class is screaming out for a hierarchical makeover. A reasonable approach would be to narrow the scope of the `SpreadsheetCell` to cover only strings, perhaps renaming it `StringSpreadsheetCell` in the process. To handle doubles, a second class, `DoubleSpreadsheetCell`, would inherit from the `StringSpreadsheetCell` and provide functionality specific to its own format. Figure 9-5 illustrates such a design. This approach models inheritance for reuse since the `DoubleSpreadsheetCell` would only be deriving from `StringSpreadsheetCell` to make use of some of its built-in functionality.

If you were to implement the design shown in Figure 9-5, you might discover that the derived class would override most, if not all, of the functionality of the base class. Since doubles are treated differently from strings in almost all cases, the relationship may not be quite as it was originally understood. Yet, there clearly is a relationship between a cell containing strings and a cell containing doubles. Rather than using the model in Figure 9-5, which implies that somehow a `DoubleSpreadsheetCell` “is-a” `StringSpreadsheetCell`, a better design would make these classes peers with a common parent, `SpreadsheetCell`. Such a design is shown in Figure 9-6.

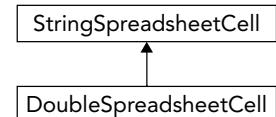


FIGURE 9-5

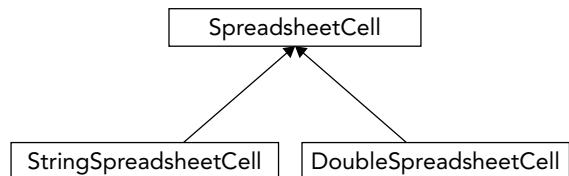


FIGURE 9-6

The design in Figure 9-6 shows a polymorphic approach to the `SpreadsheetCell` hierarchy. Since `DoubleSpreadsheetCell` and `StringSpreadsheetCell` both inherit from a common parent, `SpreadsheetCell`, they are interchangeable in the view of other code. In practical terms, that means:

- Both derived classes support the same interface (set of methods) defined by the base class.
- Code that makes use of `SpreadsheetCell` objects can call any method in the interface without even knowing whether the cell is a `DoubleSpreadsheetCell` or a `StringSpreadsheetCell`.
- Through the magic of virtual methods, the appropriate instance of every method in the interface will be called depending on the class of the object.
- Other data structures, such as the `Spreadsheet` class described in Chapter 8, can contain a collection of multityped cells by referring to the parent type.

The Spreadsheet Cell Base Class

Since all spreadsheet cells are deriving from the `SpreadsheetCell` base class, it is probably a good idea to write that class first. When designing a base class, you need to consider how the derived

classes relate to each other. From this information, you can derive the commonality that will go inside the parent class. For example, `string` cells and `double` cells are similar in that they both contain a single piece of data. Since the data is coming from the user and will be displayed back to the user, the value is set as a `string` and retrieved as a `string`. These behaviors are the shared functionality that will make up the base class.

A First Attempt

The `SpreadsheetCell` base class is responsible for defining the behaviors that all `SpreadsheetCell` derived classes will support. In this example, all cells need to be able to set their value as a `string`. All cells also need to be able to return their current value as a `string`. The base class definition declares these methods, but note that it has no data members. The reason will be explained afterwards.

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    virtual ~SpreadsheetCell();
    virtual void set(const std::string& inString);
    virtual std::string getString() const;
};
```

When you start writing the `.cpp` file for this class, you very quickly run into a problem. Since the base class of spreadsheet cell contains neither a `double` nor a `string`, how can you implement it? More generally, how do you write a parent class that declares the behaviors that are supported by derived classes without actually defining the implementation of those behaviors?

One possible approach is to implement “do nothing” functionality for those behaviors. For example, calling the `set()` method on the `SpreadsheetCell` base class will have no effect because the base class has nothing to set. This approach still doesn’t feel right, however. Ideally, there should never be an object that is an instance of the base class. Calling `set()` should always have an effect because it should always be called on either a `DoubleSpreadsheetCell` or a `StringSpreadsheetCell`. A good solution will enforce this constraint.

The preceding code declares a virtual destructor for the `SpreadsheetCell` class. If you don’t do this, the compiler will generate a default destructor for you but a non-virtual one. That means if you don’t declare the virtual destructor yourself, you might encounter problems with destruction of derived classes through pointers or references as explained earlier in this chapter.

Pure Virtual Methods and Abstract Base Classes

Pure virtual methods are methods that are explicitly undefined in the class definition. By making a method pure virtual, you are telling the compiler that no definition for the method exists in the current class. Thus, the class is said to be *abstract* because no other code will be able to instantiate it. The compiler enforces the fact that if a class contains one or more pure virtual methods, it can never be used to construct an object of that type.

There is a special syntax for designating a pure virtual method. The method declaration is followed by `=0`. No implementation needs to be written.

```

class SpreadsheetCell
{
public:
    SpreadsheetCell();
    virtual ~SpreadsheetCell();
    virtual void set(const std::string& inString) = 0;
    virtual std::string getString() const = 0;
};

```

Now that the base class is an abstract class, it is impossible to create a `SpreadsheetCell` object. The following code will not compile, and will give an error such as `Cannot declare object of type 'SpreadsheetCell'` because one or more virtual functions are abstract:

```
SpreadsheetCell cell; // Error! Attempts creating abstract class instance
```

However, the following code will compile:

```
SpreadsheetCell* ptr = nullptr;
```

This works because it will require instantiating a derived class of the abstract base class, for example:

```
ptr = new StringSpreadsheetCell();
```

NOTE *An abstract class provides a way to prevent other code from instantiating an object directly, as opposed to one of its derived classes.*

Base Class Source Code

There is not much code required for `SpreadsheetCell.cpp`. As the class was defined, most of the methods are pure virtual — there is no definition to give. All that is left is the constructor and destructor. For this example, the constructor is implemented just as a placeholder in case initialization needs to happen in the future. However, the virtual destructor is required because this is a base class.

```
SpreadsheetCell::SpreadsheetCell() { }
SpreadsheetCell::~SpreadsheetCell() { }
```

The Individual Derived Classes

Writing the `StringSpreadsheetCell` and `DoubleSpreadsheetCell` classes is just a matter of implementing the functionality that is *defined* in the parent. Because you want clients to be able to instantiate and work with `string` cells and `double` cells, the cells can't be abstract — they *must* implement all of the pure virtual methods inherited from their parent. If a derived class does not implement all pure virtual methods from the base class, then the derived class will be abstract as well, and clients will not be able to instantiate objects of the derived class.

String Spreadsheet Cell Class Definition

The first step in writing the class definition of `StringSpreadsheetCell` is to inherit from `SpreadsheetCell`:

```
class StringSpreadsheetCell : public SpreadsheetCell
{
```

You want to initialize the data value of the `StringSpreadsheetCell` to “#NOVALUE” to indicate that the value has not been set. The compiler generated default constructor will call the default `string` constructor for `mValue` which will set the initial value to the empty string, “”. So, you need to provide an explicit default constructor and initialize the value yourself.

```
public:
    StringSpreadsheetCell();
```

Next, the inherited pure virtual methods are overridden, this time without being set to zero:

```
virtual void set(const std::string& inString) override;
virtual std::string getString() const override;
```

Finally, the `string` cell adds a private data member, `mValue`, which stores the actual cell data:

```
private:
    std::string mValue;
};
```

String Spreadsheet Cell Implementation

The `.cpp` file for `StringSpreadsheetCell` is a bit more interesting than the base class. In the constructor, `mValue` is initialized to a `string` that indicates that no value has been set.

```
StringSpreadsheetCell::StringSpreadsheetCell() : mValue("#NOVALUE") {}
```

The `set` method is straightforward since the internal representation is already a `string`. Similarly, the `getString()` method returns the stored value.

```
void StringSpreadsheetCell::set(const string& inString)
{
    mValue = inString;
}
string StringSpreadsheetCell::getString() const
{
    return mValue;
}
```

Double Spreadsheet Cell Class Definition and Implementation

The double version follows a similar pattern, but with different logic. In addition to the `set()` method that takes a `string`, it also provides a new `set()` method that allows a client to set the value with a `double`. Two new `private` methods are used to convert between a `string` and a `double` and vice versa. As in `StringSpreadsheetCell`, it has a data member called `mValue`, this time a `double`.

```
class DoubleSpreadsheetCell : public SpreadsheetCell
{
public:
    DoubleSpreadsheetCell();
    virtual void set(double inDouble);
    virtual void set(const std::string& inString) override;
    virtual std::string getString() const override;
private:
    static std::string doubleToString(double inValue);
    static double stringToDouble(const std::string& inValue);
    double mValue;
};
```

The implementation of the `DoubleSpreadsheetCell` constructor is as follows.

```
DoubleSpreadsheetCell::DoubleSpreadsheetCell()
: mValue(std::numeric_limits<double>::quiet_NaN())
{}
```

NOTE *This implementation initializes `mValue` to `std::numeric_limits<double>::quiet_NaN()` where `Nan` stands for “Not a Number.” You need to include `<limits>` to use `std::numeric_limits<>`.*

The `set()` method that takes a `double` is straightforward. The `string` version uses the `private static` method `stringToDouble()`. The `getString()` method converts the stored `double` value into a `string`:

```
void DoubleSpreadsheetCell::set(double inDouble)
{
    mValue = inDouble;
}
void DoubleSpreadsheetCell::set(const string& inString)
{
    mValue = stringToDouble(inString);
}
string DoubleSpreadsheetCell::getString() const
{
    return doubleToString(mValue);
}
```

You may already see one major advantage of implementing spreadsheet cells in a hierarchy — the code is much simpler. You don't need to worry about using two fields to represent the two types of data. Each object can be self-centered and only deal with its own functionality.

Note that the implementations of `doubleToString()` and `stringToDouble()` are omitted because they are the same as in Chapter 7.

Leveraging Polymorphism

Now that the `SpreadsheetCell` hierarchy is polymorphic, client code can take advantage of the many benefits that polymorphism has to offer. The following test program explores many of these features:

```
int main()
{
```

To demonstrate polymorphism, this test program declares a vector of three `SpreadsheetCell` pointers. Remember that since `SpreadsheetCell` is an abstract class, you can't create objects of that type. However, you can still have a pointer or reference to a `SpreadsheetCell` because it would actually be pointing to one of the derived classes. This vector, because it is an vector of the parent type `SpreadsheetCell`, allows you to store a heterogeneous mixture of the two derived classes. This means that elements of the vector could be either a `StringSpreadsheetCell` or a `DoubleSpreadsheetCell`.

```
vector<unique_ptr<SpreadsheetCell>> cellArray;
```

The first element of the vector is set to point to a new `StringSpreadsheetCell`; the second is also set to a new `StringSpreadsheetCell`, and the third is a new `DoubleSpreadsheetCell`.

```
cellArray.push_back(make_unique<StringSpreadsheetCell>());
cellArray.push_back(make_unique<StringSpreadsheetCell>());
cellArray.push_back(make_unique<DoubleSpreadsheetCell>());
```

Now that the vector contains multityped data, any of the methods declared by the base class can be applied to the objects in the vector. The code just uses `SpreadsheetCell` pointers — the compiler has no idea at compile time what types the objects actually are. However, because they are inheriting from `SpreadsheetCell`, they must support the methods of `SpreadsheetCell`.

```
cellArray[0] ->set("hello");
cellArray[1] ->set("10");
cellArray[2] ->set("18");
```

When the `getString()` method is called, each object properly returns a `string` representation of their value. The important, and somewhat amazing, thing to realize is that the different objects do this in different ways. A `StringSpreadsheetCell` will return its stored value. A `DoubleSpreadsheetCell` will first perform a conversion. As the programmer, you don't need to know how the object does it — you just need to know that because the object is a `SpreadsheetCell`, it *can* perform this behavior.

```

        cout << "Vector values are [" << cellArray[0]->getString() << ","
            cellArray[1]->getString() << ","
            cellArray[2]->getString() << "]"
            << endl;
    return 0;
}

```

Future Considerations

The new implementation of the `SpreadsheetCell` hierarchy is certainly an improvement from an object-oriented design point of view. Yet, it would probably not suffice as an actual class hierarchy for a real-world spreadsheet program for several reasons.

First, despite the improved design, one feature of the original is still missing: the ability to convert from one cell type to another. By dividing them into two classes, the cell objects become more loosely integrated. To provide the ability to convert from a `DoubleSpreadsheetCell` to a `StringSpreadsheetCell`, you could add a typed constructor. It would have a similar appearance to a copy constructor but instead of a reference to an object of the same class, it would take a reference to an object of a sibling class.

```

class StringSpreadsheetCell : public SpreadsheetCell
{
public:
    StringSpreadsheetCell();
    StringSpreadsheetCell(const DoubleSpreadsheetCell& inDoubleCell);
    // Omitted for brevity
};

```

With a typed constructor, you can easily create a `StringSpreadsheetCell` given a `DoubleSpreadsheetCell`. Don't confuse this with casting, however. Casting from one sibling to another will not work, unless you overload the cast operator as described in Chapter 14.

WARNING *You can always cast up the hierarchy, and you can sometimes cast down the hierarchy. Casting across the hierarchy is possible by changing the behavior of the cast operator, or by using reinterpret_cast<>, which is not recommended.*

The question of how to implement overloaded operators for cells is an interesting one, and there are several possible solutions. One approach is to implement a version of each operator for every combination of cells. With only two derived classes, this is manageable. There would be an `operator+` function to add two double cells, to add two string cells, and to add a double cell to a string cell. Another approach is to decide on a common representation. The preceding implementation already standardizes on a string as a common representation of sorts. A single `operator+` function could cover all the cases by taking advantage of this common representation. One

possible implementation, which assumes that the result of adding two cells is always a `string` cell, is as follows:

```
StringSpreadsheetCell operator+(const StringSpreadsheetCell& lhs,
                                const StringSpreadsheetCell& rhs)
{
    StringSpreadsheetCell newCell;
    newCell.set(lhs.getString() + rhs.getString());
    return newCell;
}
```

As long as the compiler has a way to turn a particular cell into a `StringSpreadsheetCell`, the operator will work. Given the previous example of having a `StringSpreadsheetCell` constructor that takes a `DoubleSpreadsheetCell` as an argument, the compiler will automatically perform the conversion if it is the only way to get the `operator+` to work. That means the following code will work, even though `operator+` was explicitly written to work on `StringSpreadsheetCells`:

```
DoubleSpreadsheetCell myDbl;
myDbl.set(8.4);
StringSpreadsheetCell result = myDbl + myDbl;
```

Of course, the result of this addition won't really add the numbers together. It will convert the double cells into `string` cells and add the strings, resulting in a `StringSpreadsheetCell` with a value of `8.48.4`.

If you are still feeling a little unsure about polymorphism, start with the code for this example and try things out. The `main()` function in the preceding example is a great starting point for experimental code that simply exercises various aspects of the class.

MULTIPLE INHERITANCE

As you read in Chapter 5, multiple inheritance is often perceived as a complicated and unnecessary part of object-oriented programming. I'll leave the decision of whether or not it is useful up to you and your coworkers. This section explains the mechanics of multiple inheritance in C++.

Inheriting from Multiple Classes

Defining a class to have multiple parent classes is very simple from a syntactic point of view. All you need to do is list the base classes individually when declaring the class name.

```
class Baz : public Foo, public Bar
{
    // Etc.
};
```

By listing multiple parents, the `Baz` object will have the following characteristics:

- A `Baz` object will support the `public` methods and contain the data members of both `Foo` and `Bar`.
- The methods of the `Baz` class will have access to `protected` data and methods in both `Foo` and `Bar`.
- A `Baz` object can be upcast to either a `Foo` or a `Bar`.
- Creating a new `Baz` object will automatically call the `Foo` and `Bar` default constructors, in the order that the classes are listed in the class definition.
- Deleting a `Baz` object will automatically call the destructors for the `Foo` and `Bar` classes, in the reverse order that the classes are listed in the class definition.

The following example shows a class, `DogBird`, that has two parent classes — a `Dog` class and a `Bird` class. The fact that a dog-bird is a ridiculous example should not be viewed as a statement that multiple inheritance itself is ridiculous. Honestly, I leave that judgment up to you.

```
class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
};

class DogBird : public Dog, public Bird
{
```

The class hierarchy for `DogBird` is shown in Figure 9-7.

Using objects of classes with multiple parents is no different from using objects without multiple parents. In fact, the client code doesn't even have to know that the class has two parents. All that really matters are the properties and behaviors supported by the class. In this case, a `DogBird` object supports all of the `public` methods of `Dog` and `Bird`.

```
DogBird myConfusedAnimal;
myConfusedAnimal.bark();
myConfusedAnimal.chirp();
```

The output of this program is as follows:

```
Woof!
Chirp!
```

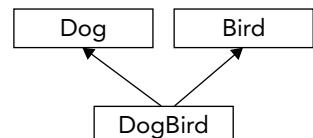


FIGURE 9-7

Naming Collisions and Ambiguous Base Classes

It's not difficult to construct a scenario where multiple inheritance would seem to break down. The following examples show some of the edge cases that must be considered:

Name Ambiguity

What if the `Dog` class and the `Bird` class both had a method called `eat()`? Since `Dog` and `Bird` are not related in any way, one version of the method does not override the other — they both continue to exist in the `DogBird` derived class.

As long as client code never attempts to call the `eat()` method, that is not a problem. The `DogBird` class will compile correctly despite having two versions of `eat()`. However, if client code attempts to call the `eat()` method on a `DogBird`, the compiler will give an error indicating that the call to `eat()` is ambiguous. The compiler will not know which version to call. The following code provokes this ambiguity error:

```
class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() { cout << "The dog has eaten." << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() { cout << "The bird has eaten." << endl; }
};

class DogBird : public Dog, public Bird
{
};

int main()
{
    DogBird myConfusedAnimal;
    myConfusedAnimal.eat();    // Error! Ambiguous call to method eat()
    return 0;
}
```

The solution to the ambiguity is to either explicitly upcast the object, essentially hiding the undesired version of the method from the compiler, or to use a disambiguation syntax. For example, the following code shows two ways to invoke the `Dog` version of `eat()`:

```
dynamic_cast<Dog&>(myConfusedAnimal).eat(); // Calls Dog::eat()
myConfusedAnimal.Dog::eat();                  // Calls Dog::eat()
```

Methods of the derived class itself can also explicitly disambiguate between different methods of the same name by using the same syntax used to access parent methods, the `::` operator. For example, the `DogBird` class could prevent ambiguity errors in other code by defining its own `eat()` method. Inside this method, it would determine which parent version to call.

```
void DogBird::eat()
{
    Dog::eat(); // Explicitly call Dog's version of eat()
}
```

Yet another way to prevent the ambiguity error is to use a `using` statement to explicitly state which version of `eat()` should be inherited in `DogBird`. This is done in the following `DogBird` definition:

```
class DogBird : public Dog, public Bird
{
public:
    using Dog::eat; // Explicitly inherit Dog's version of eat()
};
```

Another way to provoke ambiguity is to inherit from the same class twice. For example, if the `Bird` class inherited from `Dog` for some reason, the code for `DogBird` would not compile because `Dog` becomes an ambiguous base class.

```
class Dog {};
class Bird : public Dog {};
class DogBird : public Bird, public Dog {}; // Error! Dog is an ambiguous base
class.
```

Most occurrences of ambiguous base classes are either contrived “what-if” examples, as in the preceding, or arise from untidy class hierarchies.

Figure 9-8 shows a class diagram for the preceding example, indicating the ambiguity.

Ambiguity can also occur with data members. If `Dog` and `Bird` both had a data member with the same name, an ambiguity error would occur when client code attempted to access that member.

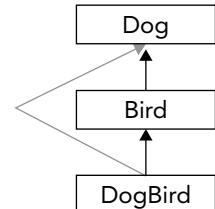


FIGURE 9-8

Ambiguous Base Classes

A more likely scenario is that multiple parents themselves have common parents. For example, perhaps both `Bird` and `Dog` are inheriting from an `Animal` class, as shown in Figure 9-9.

This type of class hierarchy is permitted in C++, though name ambiguity can still occur. For example, if the `Animal` class has a public method called `sleep()`, that method could not be called on a `DogBird` object because the compiler would not know whether to call the version inherited by `Dog` or by `Bird`.

The best way to use these “diamond-shaped” class hierarchies is to make the topmost class an abstract base class with all methods declared as pure virtual. Since the class only declares methods without providing definitions, there are no methods in the base class to call and thus there are no ambiguities at that level.

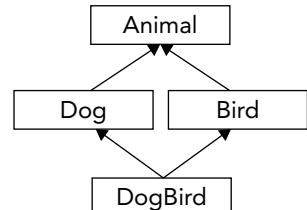


FIGURE 9-9

The following example implements a diamond-shaped class hierarchy with a pure virtual `eat()` method that must be defined by each derived class. The `DogBird` class still needs to be explicit about which parent's `eat()` method it uses, but any ambiguity would be caused by `Dog` and `Bird` having the same method, not because they inherit from the same class.

```
class Animal
{
public:
    virtual void eat() = 0;
};

class Dog : public Animal
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() override { cout << "The dog has eaten." << endl; }
};

class Bird : public Animal
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() override { cout << "The bird has eaten." << endl; }
};

class DogBird : public Dog, public Bird
{
public:
    using Dog::eat;
};
```

A more refined mechanism for dealing with the top class in a diamond-shaped hierarchy, *virtual base classes*, is explained at the end of this chapter.

Uses for Multiple Inheritance

At this point, you're probably wondering why programmers would want to tackle multiple inheritance in their code. The most straightforward use case for multiple inheritance is to define a class of objects that is-a something and also is-a something else. As was said in Chapter 5, any real-world objects you find that follow this pattern are unlikely to translate well into code.

One of the most compelling and simple uses of multiple inheritance is for the implementation of mixin classes. Mixin classes are explained in Chapter 5.

Another reason that people sometimes use multiple inheritance is to model a component-based class. Chapter 5 gave the example of an airplane simulator. The `Airplane` class has an engine, a fuselage, controls, and other components. While the typical implementation of an `Airplane` class would make each of these components a separate data member, you *could* use multiple inheritance. The airplane class would inherit from engine, fuselage, and controls, in effect getting the behaviors and properties of all of its components. I recommend you stay away from this type of code because it confuses a clear has-a relationship with inheritance, which should be used for is-a relationships. The recommended solution is to have an `Airplane` class which contains data members of type `Engine`, `Fuselage`, and `Controls`.

INTERESTING AND OBSCURE INHERITANCE ISSUES

Extending a class opens up a variety of issues. What characteristics of the class can and cannot be changed? What does the mysterious `virtual` keyword really do? These questions, and many more, are answered in the following sections.

Changing the Overridden Method's Characteristics

For the most part, the reason you override a method is to change its implementation. Sometimes, however, you may want to change other characteristics of the method.

Changing the Method Return Type

A good rule of thumb is to override a method with the exact method declaration, or *method prototype*, that the base class uses. The implementation can change, but the prototype stays the same.

That does not have to be the case, however. In C++, an overriding method can change the return type as long as the original return type is a pointer or reference to a class, and the new return type is a pointer or reference to a descendent class. Such types are called *covariant return types*. This feature sometimes comes in handy when the base class and derived class work with objects in a *parallel hierarchy*. That is, another group of classes that is tangential, but related, to the first class hierarchy.

For example, consider a hypothetical cherry orchard simulator. You might have two hierarchies of classes that model different real-world objects but are obviously related. The first is the cherry chain. The base class, `Cherry`, has a derived class called `BingCherry`. Similarly, there is another chain of classes with a base class called `CherryTree` and a derived class called `BingCherryTree`. Figure 9-10 shows the two class chains.

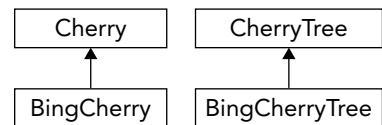


FIGURE 9-10

Now assume that the `CherryTree` class has a virtual method called `pick()` that will retrieve a single cherry from the tree:

```

Cherry* CherryTree::pick()
{
    return new Cherry();
}
  
```

In the `BingCherryTree` derived class, you may want to override this method. Perhaps Bing Cherries need to be polished when they are picked (bear with me on this one). Because a `BingCherry` is a `Cherry`, you could leave the method prototype as is and override the method as in the following example. The `BingCherry` pointer is automatically cast to a `Cherry` pointer. Note that this implementation uses a `unique_ptr` to make sure no memory is leaked when `polish()` throws an exception.

```

Cherry* BingCherryTree::pick()
{
    auto theCherry = std::make_unique<BingCherry>();
    theCherry->polish();
    return theCherry.get();
}
  
```

```
    theCherry->polish();
    return theCherry.release();
}
```

The implementation above is perfectly fine and is probably the way that I would write it. However, because you know that the `BingCherryTree` will always return `BingCherry` objects, you could indicate this fact to potential users of this class by changing the return type, as shown here:

```
BingCherry* BingCherryTree::pick()
{
    auto theCherry = std::make_unique<BingCherry>();
    theCherry->polish();
    return theCherry.release();
}
```

A good way to figure out whether you can change the return type of an overridden method is to consider whether existing code would still work, the Liskov substitution principle (LSP). In the preceding example, changing the return type was fine because any code that assumed that the `pick()` method would always return a `Cherry*` would still compile and work correctly. Because a `BingCherry` is a `Cherry`, any methods that were called on the result of `CherryTree`'s version of `pick()` could still be called on the result of `BingCherryTree`'s version of `pick()`.

You could *not*, for example, change the return type to something completely unrelated, such as `void*`. The following code will not compile:

```
void* BingCherryTree::pick() // Error!
{
    auto theCherry = std::make_unique<BingCherry>();
    theCherry->polish();
    return theCherry.release();
}
```

This will generate a compiler error, something like '`BingCherryTree::pick`': overriding virtual function return type differs and is not covariant from '`CherryTree::pick`'.

Changing the Method Parameters

If you use the name of a virtual method from the parent class in the definition of a derived class, but it uses different parameters than the method with that name uses in the parent class, it is not overriding the method of the parent class — it is creating a new method. Returning to the `Super` and `Sub` example from earlier in this chapter, you could *attempt* to override `someMethod()` in `Sub` with a new argument list as follows:

```
class Super
{
public:
    Super();
    virtual void someMethod();
};
```

```

class Sub : public Super
{
public:
    Sub();
    virtual void someMethod(int i); // Compiles, but doesn't override
    virtual void someOtherMethod();
};

```

The implementation of this method could be as follows:

```

void Sub::someMethod(int i)
{
    cout << "This is Sub's version of someMethod with argument " << i
    << "." << endl;
}

```

The preceding class definition will compile, but you have not overridden `someMethod()`. Because the arguments are different, you have created a new method that exists only in `Sub`. If you want a method called `someMethod()` that takes an `int`, and you want it to work only on objects of class `Sub`, the preceding code is correct.

In fact, the C++ standard says that the original method is now hidden as far as `Sub` is concerned. The following sample code will not compile because there is no longer a no-argument version of `someMethod()`.

```

Sub mySub;
mySub.someMethod(); // Error! Won't compile because original method is hidden.

```

If instead you really did intend to override `someMethod()` from the base class, then you should have marked the `someMethod(int)` declaration of `Sub` with the `override` specifier. By doing so, the compiler will give an error stating that `someMethod(int)` is not overriding a method from the base class.

There is a somewhat obscure technique you can use to have your cake and eat it too. That is, you can use this technique to effectively “override” a method in the derived class with a new prototype but continue to inherit the base class version. This technique uses the `using` keyword to explicitly include the base class definition of the method within the derived class as follows:

```

class Super
{
public:
    Super();
    virtual void someMethod();
};

class Sub : public Super
{
public:
    Sub();
    using Super::someMethod; // Explicitly "inherits" the Super version
    virtual void someMethod(int i); // Adds a new version of someMethod
    virtual void someOtherMethod();
};

```

NOTE It is rare to find a method in a derived class with the same name as a method in the base class but using a different parameter list.

The override Keyword

Sometimes, it is possible to accidentally create a new virtual method instead of overriding a method from the base class. Take the following `Super` and `Sub` classes where `Sub` is properly overriding `someMethod()`:

```
class Super
{
    public:
        Super();
        virtual void someMethod(double d);
};

class Sub : public Super
{
    public:
        Sub();
        virtual void someMethod(double d);
};
```

You can call `someMethod()` through a reference as follows:

```
Sub mySub;
Super& ref = mySub;
ref.someMethod(1.1); // Calls Sub's version of someMethod()
```

This will correctly call the overridden `someMethod()` from the `Sub` class. Now, suppose you used an integer parameter instead of a double while overriding `someMethod()` as follows:

```
class Sub : public Super
{
    public:
        Sub();
        virtual void someMethod(int i);
};
```

As seen in the previous section, this code will **not** override `someMethod()`, but will instead create a new virtual method. If you try to call `someMethod()` through a reference as in the following code, `someMethod()` of `Super` will be called instead of the one from `Sub`.

```
Sub mySub;
Super& ref = mySub;
ref.someMethod(1.1); // Calls Super's version of someMethod()
```

This type of problem can happen when you start to modify the `Super` class but forget to update all derived classes. For example, maybe your first version of the `Super` class had a method called `someMethod()` accepting an `integer`. You then wrote the `Sub` derived class overriding this `someMethod()` accepting an `integer`. Later you decide that `someMethod()` in the `Super` class needs a `double` instead of an `integer`, so you update `someMethod()` in the `Super` class. It might happen that at that time, you forget to update the `someMethod()` methods in derived classes to also accept a `double` instead of an `integer`. By forgetting this, you are now actually creating a new `virtual` method instead of properly overriding the method.

You can prevent this situation by using the `override` keyword as follows:

```
class Sub : public Super
{
public:
    Sub();
    virtual void someMethod(int i) override;
};
```

This definition of `Sub` will generate a compiler error, because with the `override` keyword you are saying that `someMethod()` is supposed to be overriding a method from the `Super` class, but in the `Super` class there is no `someMethod()` accepting an `integer`, only one accepting a `double`.

The problem of accidentally creating a new method instead of properly overriding one can also happen when you rename a method in the base class, and forget to rename the overriding methods in derived classes.

The `override` keyword has already been used throughout the examples in this chapter and it is recommended that you use it whenever you want to override a base class method.

Inherited Constructors

In an earlier section you saw the use of the `using` keyword to explicitly include the base class definition of a method within a derived class. This works perfectly for normal class methods, but it also works for constructors allowing you to inherit constructors from your base classes. Take a look at the following definition for the `Super` and `Sub` classes:

```
class Super
{
public:
    Super(const std::string& str);
};

class Sub : public Super
{
public:
    Sub(int i);
};
```

You can construct a `Super` object only with the provided `Super` constructor, which requires a `string` as argument. On the other hand, constructing a `Sub` object can happen only with the provided `Sub`

constructor, which requires a single integer as argument. You cannot construct `Sub` objects by using the constructor accepting a string defined in the `Super` class. For example:

```
Super super("Hello"); // OK, calls string based Super ctor
Sub sub1(1);          // OK, calls integer based Sub ctor
Sub sub2("Hello");    // Error, Sub does not inherit string Super ctor
```

If you would like the ability to construct `Sub` objects using the string based `Super` constructor, you can explicitly inherit the `Super` constructors in the `Sub` class as follows:

```
class Sub : public Super
{
public:
    using Super::Super;
    Sub(int i);
};
```

Now you can construct `Sub` objects in the following two ways:

```
Sub sub1(1);          // OK, calls integer based Sub ctor
Sub sub2("Hello");    // OK, calls inherited string based Super ctor
```

The `Sub` class can define a constructor with the same parameter list as one of the inherited constructors in the `Super` class. In this case, as with any override, the constructor of the `Sub` class takes precedence over the inherited constructor. In the following example, the `Sub` class inherits all constructors from the `Super` class with the `using` keyword. However, since the `Sub` class defines its own constructor with a single argument of type `float`, the inherited constructor from the `Super` class with a single argument of type `float` is overridden.

```
class Super
{
public:
    Super(const std::string& str);
    Super(float f);
};

class Sub : public Super
{
public:
    using Super::Super;
    Sub(float f);           // Overrides inherited float based Super ctor
};
```

With this definition, objects of `Sub` can be created as follows:

```
Sub sub1("Hello");    // OK, calls inherited string based Super ctor
Sub sub2(1.23f);      // OK, calls float based Sub ctor
```

A few restrictions apply to inheriting constructors from a base class with the `using` clause. When you inherit a constructor from a base class, you inherit all of them. It is not possible to inherit only a subset of the constructors of a base class. A second restriction is related to multiple inheritance. It's not possible to inherit constructors from one of the base classes if another base class has a

constructor with the same parameter list, because this will lead to ambiguity. To resolve this, the `Sub` class needs to explicitly define the conflicting constructors. For example, the following `Sub` class tries to inherit all constructors from both the `Super1` and `Super2` base classes, which results in a compiler error due to ambiguity of the `float` based constructors.

```
class Super1
{
public:
    Super1(float f);
};

class Super2
{
public:
    Super2(const std::string& str);
    Super2(float f);
};

class Sub : public Super1, public Super2
{
public:
    using Super1::Super1;
    using Super2::Super2;
    Sub(char c);
};
```

The first `using` clause in `Sub` inherits the constructors from `Super1`. This means that `Sub` will get the following constructor:

```
Sub(float f); // Inherited from Super1
```

With the second `using` clause in `Sub` you are trying to inherit all constructors from `Super2`. However, this will generate a compiler error because this would mean that `Sub` gets a second `Sub(float f)` constructor. The problem is solved by explicitly declaring conflicting constructors in the `Sub` class as follows:

```
class Sub : public Super1, public Super2
{
public:
    using Super1::Super1;
    using Super2::Super2;
    Sub(char c);
    Sub(float f);
};
```

The `Sub` class now explicitly declares a constructor with a single argument of type `float`, solving the ambiguity. If you want, this explicitly declared constructor in the `Sub` class accepting a `float` argument can still forward the call to the `Super1` and `Super2` constructors in its ctor-initializer as follows:

```
Sub::Sub(float f) : Super1(f), Super2(f) {}
```

When using inherited constructors, make sure that all member variables are properly initialized. For example, take the following new definitions for the `Super` and `Sub` classes. This example does not properly initialize the `mInt` data member in all cases which is a serious error.

```
class Super
{
public:
    Super(const std::string& str) : mStr(str) {}
private:
    std::string mStr;
};

class Sub : public Super
{
public:
    using Super::Super;
    Sub(int i) : Super(""), mInt(i) {}
private:
    int mInt;
};
```

You can create a `Sub` object as follows:

```
Sub s1(2);
```

This will call the `Sub(int i)` constructor, which will initialize the `mInt` data member of the `Sub` class and call the `Super` constructor with an empty string to initialize the `mStr` data member.

Because the `Super` constructor is inherited in the `Sub` class, you can also construct a `Sub` object as follows:

```
Sub s2("Hello World");
```

This will call the inherited `Super` constructor in the `Sub` class. However, this inherited `Super` constructor only initializes the `mStr` member variable of the `Super` class and does not initialize the `mInt` member variable of the `Sub` class, leaving it in an uninitialized state. This is usually not recommended.

The solution is to use in-class member initializers, which are discussed in Chapter 7. The following code uses an in-class member initializer to initialize `mInt` to 0. The `Sub(int i)` constructor can still change this initialization and initialize `mInt` to the constructor parameter `i`.

```
class Sub : public Super
{
public:
    using Super::Super;
    Sub(int i) : Super(""), mInt(i) {}
private:
    int mInt = 0;
};
```

Special Cases in Overriding Methods

Several special cases require attention when overriding a method. This section outlines the cases that you are likely to encounter.

The Base Class Method Is static

In C++, you cannot override a `static` method. For the most part, that's all you need to know. There are, however, a few corollaries that you need to understand.

First of all, a method cannot be both `static` and `virtual`. This is the first clue that attempting to override a `static` method will not do what you intend for it to do. If you have a `static` method in your derived class with the same name as a `static` method in your base class, you actually have two separate methods.

The following code shows two classes that both happen to have `static` methods called `beStatic()`. These two methods are in no way related.

```
class SuperStatic
{
public:
    static void beStatic() {
        cout << "SuperStatic being static." << endl; }
};

class SubStatic : public SuperStatic
{
public:
    static void beStatic() {
        cout << "SubStatic keepin' it static." << endl; }
};
```

Because a `static` method belongs to its class, calling the identically named methods on the two different classes will call their respective methods:

```
SuperStatic::beStatic();
SubStatic::beStatic();
```

This will output:

```
SuperStatic being static.
SubStatic keepin' it static.
```

Everything makes perfect sense as long as the methods are accessed by class. The behavior is less clear when objects are involved. In C++, you can call a `static` method using an object, but since the method is `static`, it has no `this` pointer and no access to the object itself, so it is equivalent to calling it by `classname::method()`. Referring to the previous example classes, you can write code as follows, but the results may be surprising.

```
SubStatic mySubStatic;
SuperStatic& ref = mySubStatic;
mySubStatic.beStatic();
ref.beStatic();
```

The first call to `beStatic()` will obviously call the `SubStatic` version because it is explicitly called on an object declared as a `SubStatic`. The second call might not work as you expect. The object is a `SuperStatic` reference, but it refers to a `SubStatic` object. In this case, `SuperStatic`'s version

of `beStatic()` will be called. The reason is that C++ doesn't care what the object actually is when calling a static method. It only cares about the compile-time type. In this case, the type is a reference to a `SuperStatic`.

The output of the previous example is as follows:

```
SubStatic keepin' it static.
SuperStatic being static.
```

NOTE static methods are scoped by the name of the class in which they are defined, but are not methods that apply to a specific object. A method in a class that calls a static method calls the version determined by normal name resolution. When called syntactically by using an object, the object is not actually involved in the call, except to determine the type at compile time.

The Base Class Method Is Overloaded

When you override a method by specifying a name and a set of parameters, the compiler implicitly hides all other instances of the name in the base class. The idea is that if you have overridden one method of a given name, you might have intended to override all the methods of that name, but simply forgot, and therefore this should be treated as an error. It makes sense if you think about it — why would you want to change some versions of a method and not others? Consider the following `sub` class, which overrides a method without overriding its associated overloaded siblings:

```
class Super
{
public:
    virtual void overload() { cout << "Super's overload()" << endl; }
    virtual void overload(int i) {
        cout << "Super's overload(int i)" << endl; }
};

class Sub : public Super
{
public:
    virtual void overload() override { cout << "Sub's overload()" << endl; }
};
```

If you attempt to call the version of `overload()` that takes an `int` parameter on a `Sub` object, your code will not compile because it was not explicitly overridden.

```
mySub.overload(2); // Error! No matching method for overload(int).
```

It is possible, however, to access this version of the method from a `Sub` object. All you need is a pointer or a reference to a `Super` object.

```
Sub mySub;
Super* ptr = &mySub;
ptr->overload(7);
```

The hiding of unimplemented overloaded methods is only skin deep in C++. Objects that are explicitly declared as instances of the subtype will not make the method available, but a simple cast to the base class will bring it right back.

The `using` keyword can be employed to save you the trouble of overloading all the versions when you really only want to change one. In the following code, the `Sub` class definition uses one version of `overload()` from `Super` and explicitly overloads the other:

```
class Super
{
public:
    virtual void overload() { cout << "Super's overload()" << endl; }
    virtual void overload(int i) {
        cout << "Super's overload(int i)" << endl; }
};

class Sub : public Super
{
public:
    using Super::overload;
    virtual void overload() override { cout << "Sub's overload()" << endl; }
};
```

The `using` clause has certain risks. Suppose a third `overload()` method is added to `Super`, which should have been overridden in `Sub`. This will now not be detected as an error because due to the `using` clause, the designer of the `Sub` class has explicitly said “I am willing to accept all other overloads of this method from the parent class.”

WARNING *To avoid obscure bugs, you should override all versions of an overloaded method, either explicitly or with the using keyword, but keep the risks of the using clause in mind.*

The Base Class Method Is private or protected

There's absolutely nothing wrong with overriding a `private` or `protected` method. Remember that the access specifier for a method determines who is able to *call* the method. Just because a derived class can't call its parent's `private` methods doesn't mean it can't override them. In fact, overriding a `private` or `protected` method is a common pattern in C++. It allows derived classes to define their own “uniqueness” that is referenced in the base class. Note that for example Java and C# only allow overriding `public` and `protected` methods, not `private` methods.

For example, the following class is part of a car simulator that estimates the number of miles the car can travel based on its gas mileage and amount of fuel left:

```
class MilesEstimator
{
public:
    virtual int getMilesLeft() const {
        return getMilesPerGallon() * getGallonsLeft();
    }
    virtual void setGallonsLeft(int inValue) { mGallonsLeft = inValue; }
```

```

        virtual int getGallonsLeft() const { return mGallonsLeft; }
private:
    int mGallonsLeft;
    virtual int getMilesPerGallon() const { return 20; }
};

```

The `getMilesLeft()` method performs a calculation based on the results of two of its own methods. The following code uses the `MilesEstimator` to calculate how many miles can be traveled with 2 gallons of gas:

```

MilesEstimator myMilesEstimator;
myMilesEstimator.setGallonsLeft(2);
cout << "I can go " << myMilesEstimator.getMilesLeft() << " more miles." << endl;

```

The output of this code is as follows:

```
I can go 40 more miles.
```

To make the simulator more interesting, you may want to introduce different types of vehicles, perhaps a more efficient car. The existing `MilesEstimator` assumes that all cars get 20 miles per gallon, but this value is returned from a separate method specifically so that a derived class could override it. Such a derived class is shown here:

```

class EfficientCarMilesEstimator : public MilesEstimator
{
private:
    virtual int getMilesPerGallon() const override { return 35; }
};

```

By overriding this one private method, the new class completely changes the behavior of existing, unmodified, public methods. The `getMilesLeft()` method in the base class will automatically call the overridden version of the private `getMilesPerGallon()` method. An example using the new class is as follows:

```

EfficientCarMilesEstimator myEstimator;
myEstimator.setGallonsLeft(2);
cout << "I can go " << myEstimator.getMilesLeft() << " more miles." << endl;

```

This time, the output reflects the overridden functionality:

```
I can go 70 more miles.
```

NOTE *Overriding private and protected methods is a good way to change certain features of a class without a major overhaul.*

The Base Class Method Has Default Arguments

Derived classes and base classes can each have different default arguments, but the argument that is used depends on the declared type of the variable, not the underlying object. Following is a simple example of a derived class that provides a different default argument in an overridden method:

```

class Super
{
public:
    virtual void go(int i = 2) {
        cout << "Super's go with i=" << i << endl;
    };
class Sub : public Super
{
public:
    virtual void go(int i = 7) override {
        cout << "Sub's go with i=" << i << endl;
    };
}

```

If `go()` is called on a `Sub` object, `Sub`'s version of `go()` will be executed with the default argument of 7. If `go()` is called on a `Super` object, `Super`'s version of `go()` will be executed with the default argument of 2. However (this is the weird part), if `go()` is called on a `Super` pointer or `Super` reference that really points to a `Sub` object, `Sub`'s version of `go()` will be called but it will use the default `Super` argument of 2. This behavior is shown in the following example:

```

Super mySuper;
Sub mySub;
Super& mySuperReferenceToSub = mySub;
mySuper.go();
mySub.go();
mySuperReferenceToSub.go();

```

The output of this code is as follows:

```

Super's go with i=2
Sub's go with i=7
Sub's go with i=2

```

The reason for this behavior is that C++ binds default arguments at compile time, thus to the type of the expression describing the object being involved, not by the actual object type. Default arguments are not “inherited” in C++. If the `Sub` class above failed to provide a default argument as its parent did, it would be overloading the `go()` method with a new non zero-argument version.

NOTE *When overriding a method that has a default argument, you should provide a default argument as well, and it should probably be the same value. It is recommended to use a symbolic constant for default values so that the same symbolic constant can be used in derived classes.*

The Base Class Method Has a Different Access Level

There are two ways you may wish to change the access level of a method — you could try to make it more restrictive or less restrictive. Neither case makes much sense in C++, but there are a few legitimate reasons for attempting to do so.

To enforce tighter restriction on a method (or on a data member for that matter), there are two approaches you can take. One way is to change the access specifier for the entire base class. This approach is described later in this chapter. The other approach is simply to redefine the access in the derived class, as illustrated in the `Shy` class that follows:

```
class Gregarious
{
    public:
        virtual void talk() override {
            cout << "Gregarious says hi!" << endl; }
};

class Shy : public Gregarious
{
    protected:
        virtual void talk() override {
            cout << "Shy reluctantly says hello." << endl; }
};
```

The `protected` version of `talk()` in the `Shy` class properly overrides the method. Any client code that attempts to call `talk()` on a `Shy` object will get a compile error:

```
myShy.talk(); // Error! Attempt to access protected method.
```

However, the method is not fully protected. One only has to obtain a `Gregarious` reference or pointer to access the method that you thought was protected:

```
Shy myShy;
Gregarious& ref = myShy;
ref.talk();
```

The output of the preceding code is: `Shy reluctantly says hello.`

This proves that making the method `protected` in the derived class did actually override the method (because the derived class version was correctly called), but it also proves that the `protected` access can't be fully enforced if the base class makes it `public`.

NOTE *There is no reasonable way (or good reason) to restrict access to a public parent method.*

NOTE *The previous example redefined the method in the derived class because it wants to display a different message. If you don't want to change the implementation, but instead only want to change the access level of a method, the preferred way is to simply add a `using` statement in the derived class definition with the desired access level.*

It's much easier (and makes a lot more sense) to lessen access restrictions in derived classes. The simplest way is simply to provide a `public` method that calls a `protected` method from the base class, as shown here:

```
class Secret
{
protected:
    virtual void dontTell() { cout << "I'll never tell." << endl; }
};
class Blabber : public Secret
{
public:
    virtual void tell() { dontTell(); }
};
```

A client calling the `public tell()` method of a `Blabber` object would effectively access the `protected` method of the `Secret` class. Of course, this doesn't *really* change the access level of `dontTell()`, it just provides a `public` way of accessing it.

You could also override `dontTell()` explicitly in `Blabber` and give it new behavior with `public` access. This makes a lot more sense than reducing the level of access because it is entirely clear what happens with a reference or pointer to the base class. For example, suppose that `Blabber` actually made the `dontTell()` method `public`:

```
class Secret
{
protected:
    virtual void dontTell() { cout << "I'll never tell." << endl; }
};
class Blabber : public Secret
{
public:
    virtual void dontTell() override { cout << "I'll tell all!" << endl; }
};
```

If the `dontTell()` method is called on a `Blabber` object, it will output `I'll tell all!`

```
myBlabber.dontTell(); // Outputs "I'll tell all!"
```

In this case, however, the `protected` method in the base class stays `protected` because any attempts to call `Secret`'s `dontTell()` method through a pointer or reference will not compile.

```
Blabber myBlabber;
Secret& ref = myBlabber;
Secret* ptr = &myBlabber;
ref.dontTell(); // Error! Attempt to access protected method.
ptr->dontTell(); // Error! Attempt to access protected method.
```

NOTE *The only truly useful way to change a method's access level is by providing a less restrictive accessor to a protected method.*

Copy Constructors and Assignment Operator in Derived Classes

Chapter 8 says that providing a copy constructor and assignment operator is considered a good programming practice when you have dynamically allocated memory in the class. When defining a derived class, you need to be careful about copy constructors and `operator=`.

If your derived class does not have any special data (pointers, usually) that require a nondefault copy constructor or `operator=`, you don't need to have one, regardless of whether or not the base class has one. If your derived class omits the copy constructor or `operator=`, a default copy constructor or `operator=` will be provided for the data members specified in the derived class and the base class copy constructor or `operator=` will be used for the data members specified in the base class.

On the other hand, if you *do* specify a copy constructor in the derived class, you need to explicitly chain to the parent copy constructor, as shown in the following code. If you do not do this, the default constructor (not the copy constructor!) will be used for the parent portion of the object.

```
class Super
{
public:
    Super();
    Super(const Super& inSuper);
};

class Sub : public Super
{
public:
    Sub();
    Sub(const Sub& inSub);
};

Sub::Sub(const Sub& inSub) : Super(inSub)
{ }
```

Similarly, if the derived class overrides `operator=`, it is almost always necessary to call the parent's version of `operator=` as well. The only case where you wouldn't do this is if there is some bizarre reason why you only want part of the object assigned when an assignment takes place. The following code shows how to call the parent's assignment operator from the derived class:

```
Sub& Sub::operator=(const Sub& inSub)
{
    if (&inSub == this) {
        return *this;
    }
    Super::operator=(inSub); // Calls parent's operator=.
    // Do necessary assignments for derived class.
    return *this;
}
```

WARNING If your derived class does not specify its own copy constructor or operator=, the parent functionality continues to work. If the derived class does provide its own copy constructor or operator=, it needs to explicitly reference the parent versions.

The Truth about virtual

When you first encountered method overriding earlier in this chapter, I told you that only `virtual` methods can be properly overridden. The reason I had to add the qualifier *properly* is that if a method is not `virtual`, you can still attempt to override it but it will be wrong in subtle ways.

Hiding Instead of Overriding

The following code shows a base class and a derived class, each with a single method. The derived class is attempting to override the method in the base class, but it is not declared to be `virtual` in the base class.

```
class Super
{
public:
    void go() { cout << "go() called on Super" << endl; }
};

class Sub : public Super
{
public:
    void go() { cout << "go() called on Sub" << endl; }
};
```

Attempting to call the `go()` method on a `Sub` object will initially appear to work.

```
Sub mySub;
mySub.go();
```

The output of this call is, as expected, `go() called on Sub`. However, since the method was not `virtual`, it was not actually overridden. Rather, the `Sub` class created a new method, also called `go()` that is completely unrelated to the `Super` class's method called `go()`. To prove this, simply call the method in the context of a `Super` pointer or reference.

```
Sub mySub;
Super& ref = mySub;
ref.go();
```

You would expect the output to be, `go() called on Sub`, but in fact, the output will be, `go() called on Super`. This is because the `ref` variable is a `Super` reference and because the `virtual` keyword was omitted. When the `go()` method is called, it simply executes `Super`'s `go()` method. Because it is not `virtual`, there is no need to consider whether a derived class has overridden it.

WARNING *Attempting to override a non-virtual method will hide the base class definition and will only be used in the context of the derived class.*

How virtual Is Implemented

To understand how method hiding is avoided, you need to know a bit more about what the `virtual` keyword actually does. When a class is compiled in C++, a binary object is created that contains all methods for the class. In the non-virtual case, the code to transfer control to the appropriate method is hard-coded directly where the method is called based on the compile-time type.

If the method is declared `virtual`, the correct implementation is called through the use of a special area of memory called the *vtable*, for “virtual table.” For each class that has one or more virtual methods there is a vtable, and every object of such a class contains a pointer to said vtable. This vtable contains pointers to the implementations of the `virtual` methods. In this way, when a method is called on an object, the pointer is followed into the vtable and the appropriate version of the method is executed based on the actual type of the object.

To better understand how vtables make overriding of methods possible, take the following `Super` and `Sub` classes as an example.

```
class Super
{
public:
    virtual void func1() {}
    virtual void func2() {}
    void nonVirtualFunc() {}
};

class Sub : public Super
{
public:
    virtual void func2() override {}
    void nonVirtualFunc() {}
};
```

For this example, assume that you have the following two instances:

```
Super mySuper;
Sub mySub;
```

Figure 9-11 shows a high-level view of how the vtables for both instances look. The `mySuper` object contains a pointer to its vtable. This vtable has two entries, one for `func1()` and one for `func2()`. Those entries point to the implementations of `Super::func1()` and `Super::func2()`.

`mySub` also contains a pointer to its vtable which also has two entries, one for `func1()` and one for `func2()`. The `func1()` entry of the `mySub` vtable points to `Super::func1()` because `Sub` does not override `func1()`. On the other hand, the `func2()` entry of the `mySub` vtable points to `Sub::func2()`.

Note that both vtables do not contain any entry for the `nonVirtualFunc()` method because that method is not virtual.

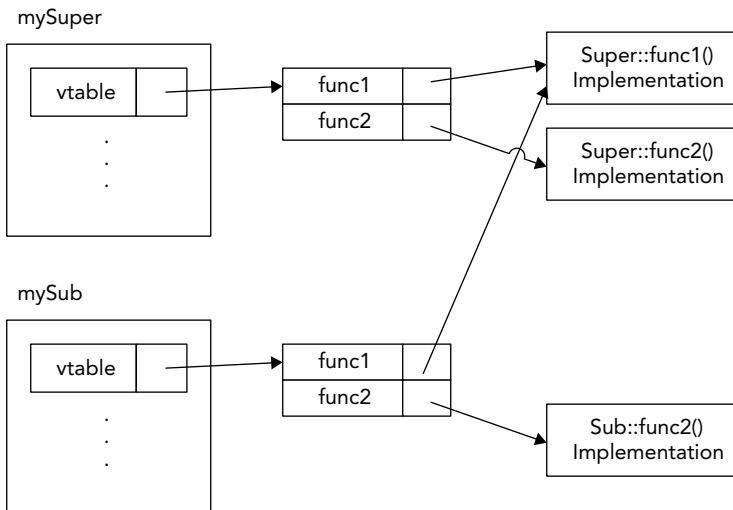


FIGURE 9-11

The Justification for `virtual`

Given the fact that you are advised to make all methods `virtual`, you might be wondering why the `virtual` keyword even exists. Can't the compiler automatically make all methods `virtual`? The answer is yes, it *could*. Many people think that the language *should* just make everything `virtual`. The Java language effectively does this.

The argument against making everything `virtual`, and the reason that the keyword was created in the first place, has to do with the overhead of the vtable. To call a `virtual` method, the program needs to perform an extra operation by dereferencing the pointer to the appropriate code to execute. This is a minuscule performance hit for most cases, but the designers of C++ thought that it was better, at least at the time, to let the programmer decide if the performance hit was necessary. If the method was never going to be overridden, there was no need to make it `virtual` and take the performance hit. However, with today's CPUs, the performance hit is measured in fractions of a nanosecond and this will keep getting smaller with future CPUs. In most applications, you will not have a measurable performance difference between using `virtual` methods and avoiding them, so you should still follow the advice of making all methods, especially destructors, `virtual`.

There is also a tiny hit to memory usage for each object. In addition to the implementation of the method, each object would also need a pointer, which takes up a tiny amount of space.

The Need for `virtual` Destructors

Even programmers who don't adopt the guideline of making all methods `virtual` still adhere to the rule when it comes to destructors. The reason is that making your destructors non-`virtual` can easily result in situations in which memory is not freed by object destruction. In only one case it is allowed for a destructor to be non-`virtual`, that is for a class that is marked as `final`.

For example, if a derived class uses memory that is dynamically allocated in the constructor and deleted in the destructor, it will never be freed if the destructor is never called. As the following code shows, it is easy to “trick” the compiler into skipping the call to the destructor if it is non-virtual:

```

class Super
{
public:
    Super() {}
    ~Super() {}

};

class Sub : public Super
{
public:
    Sub() { mString = new char[30]; }
    ~Sub() { delete [] mString; }

private:
    char* mString;
};

int main()
{
    Super* ptr = new Sub();    // mString is allocated here.
    delete ptr;    // ~Super is called, but not ~Sub because the destructor
                   // is not virtual!
    return 0;
}

```

WARNING *Unless you have a specific reason not to, or the class is marked as final, I recommend making all methods, including destructors but not constructors, virtual. Constructors cannot and need not be virtual because you always specify the exact class being constructed when creating an object.*

Run-Time Type Facilities

Relative to other object-oriented languages, C++ is very compile-time oriented. Overriding methods, as you learned above, works because of a level of indirection between a method and its implementation, not because the object has built-in knowledge of its own class.

There are, however, features in C++ that provide a run-time view of an object. These features are commonly grouped together under a feature set called *Run-Time Type Information*, or *RTTI*. RTTI provides a number of useful features for working with information about an object’s class membership. One such feature is `dynamic_cast` to safely convert between types within an OO hierarchy and is discussed earlier in this chapter.

A second RTTI feature is the `typeid` operator, which lets you query an object at run time to find out its type. For the most part, you shouldn’t ever need to use `typeid` because any code that is conditionally run based on the type of the object would be better handled with `virtual` methods.

The following code uses `typeid` to print a message based on the type of the object:

```
#include <typeinfo>
void speak(const Animal& inAnimal)
{
    if (typeid(inAnimal) == typeid(Dog)) {
        cout << "Woof!" << endl;
    } else if (typeid(inAnimal) == typeid(Bird)) {
        cout << "Chirp!" << endl;
    }
}
```

Anytime you see code like this, you should immediately consider reimplementing the functionality as a virtual method. In this case, a better implementation would be to declare a virtual method called `speak()` in the `Animal` class. `Dog` would override the method to print "Woof!" and `Bird` would override the method to print "Chirp!". This approach better fits object-oriented programming, where functionality related to objects is given to those objects.

WARNING *The `typeid` operator only works correctly if the class has at least one virtual method. Using `dynamic_cast` on a class without any virtual method will cause a compiler error.*

One of the primary values of the `typeid` operator is for logging and debugging purposes. The following code makes use of `typeid` for logging. The `logObject()` function takes a "loggable" object as a parameter. The design is such that any object that can be logged inherits from the `Loggable` class and supports a method called `getLogMessage()`. In this way, `Loggable` is a mixin class.

```
#include <typeinfo>
void logObject(const Loggable& inLoggableObject)
{
    logfile << typeid(inLoggableObject).name() << " ";
    logfile << inLoggableObject.getLogMessage() << endl;
}
```

The `logObject()` function first writes the name of the object's class to the file, followed by its log message. This way, when you read the log later, you can see which object was responsible for every line of the file.

NOTE *If you are using `typeid` other than for logging and debugging purposes, consider reimplementing it using virtual methods.*

Non-Public Inheritance

In all previous examples, parent classes were always listed using the `public` keyword. You may be wondering if a parent can be `private` or `protected`. In fact it can, though neither is as common as `public`.

Declaring the relationship with the parent to be `protected` means that all `public` methods and data members from the base class become `protected` in the context of the derived class. Similarly, specifying `private` access means that all `public` and `protected` methods and data members of the base class become `private` in the derived class.

There are a handful of reasons why you might want to uniformly degrade the access level of the parent in this way, but most reasons imply flaws in the design of the hierarchy. Some programmers abuse this language feature, often in combination with multiple inheritance, to implement “components” of a class. Instead of making an `Airplane` class that contains an `engine` data member and a `fuselage` data member, they make an `Airplane` class that is a `protected` engine and a `protected` fuselage. In this way, the `Airplane` doesn’t look like an engine or a fuselage to client code (because everything is `protected`), but it is able to use all of that functionality internally.

NOTE Non-public inheritance is rare and I recommend using it cautiously, if for no other reason than because most programmers are not familiar with it.

Virtual Base Classes

Earlier in this chapter, you learned about ambiguous base classes, a situation that arises when multiple parents each have a parent in common, as shown again in Figure 9-12. The solution that I recommended was to make sure that the shared parent doesn’t have any functionality of its own. That way, its methods can never be called and there is no ambiguity problem.

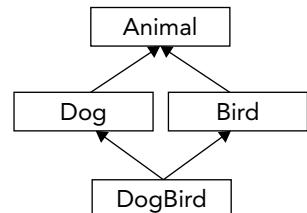


FIGURE 9-12

C++ has another mechanism for addressing this problem in the event that you do want the shared parent to have its own functionality. If the shared parent is a *virtual base class*, there will not be any ambiguity. The following code adds a `sleep()` method to the `Animal` base class and modifies the `Dog` and `Bird` classes to inherit from `Animal` as a virtual base class. Without the `virtual` keyword, a call to `sleep()` on a `DogBird` object would be ambiguous and would generate a compiler error because `DogBird` would have two subobjects of class `Animal`, one coming from `Dog` and one coming from `Bird`. However, when `Animal` is inherited virtually, `DogBird` has only one subobject of class `Animal`, so there will be no ambiguity with calling `sleep()`.

```

class Animal
{
public:
    virtual void eat() = 0;
  
```

```
        virtual void sleep() { cout << "zzzz...." << endl; }
    };
class Dog : public virtual Animal
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() override { cout << "The dog has eaten." << endl; }
};
class Bird : public virtual Animal
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() override { cout << "The bird has eaten." << endl; }
};
class DogBird : public Dog, public Bird
{
public:
    virtual void eat() override { Dog::eat(); }
};
int main()
{
    DogBird myConfusedAnimal;
    myConfusedAnimal.sleep(); // Not ambiguous because Animal is virtual
    return 0;
}
```

NOTE Virtual base classes are a great way to avoid ambiguity in class hierarchies. The only drawback is that many C++ programmers are unfamiliar with the concept.

SUMMARY

This chapter has covered numerous details about inheritance. You have learned about its many applications, including code reuse and polymorphism. You have also learned about its many abuses, including poorly-designed multiple inheritance schemes. Along the way, you've uncovered some cases that require special attention.

Inheritance is a powerful language feature that takes some time to get used to. After you have worked with the examples of this chapter and experimented on your own, I hope that inheritance will become your tool of choice for object-oriented design.

10

C++ Quirks, Oddities, and Incidentals

WHAT'S IN THIS CHAPTER?

- What the different use-cases are for references
- Keyword confusion
- How to use `typedefs` and type aliases
- What scope resolution is
- Details of features that do not fit elsewhere in this book

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Many parts of the C++ language have tricky syntax or quirky semantics. As a C++ programmer, you grow accustomed to most of this idiosyncratic behavior; it starts to feel natural. However, some aspects of C++ are a source of perennial confusion. Either books never explain them thoroughly enough, or you forget how they work and continually look them up, or both. This chapter addresses this gap by providing clear explanations for some of C++'s most niggling quirks and oddities.

Many language idiosyncrasies are covered in various chapters throughout this book. This chapter tries not to repeat those topics by limiting itself to subjects that are not covered in detail elsewhere in the book. There is a bit of redundancy with other chapters, but the material is "sliced" in a different way in order to provide you with a new perspective.

The topics of this chapter include references, `const`, `constexpr`, `static`, `extern`, `typedefs`, type aliases, casts, scope resolution, uniform initialization, initializer lists, explicit conversion operators, attributes, user-defined literals, header files, variable-length argument lists, and preprocessor macros. Although this list might appear to be a hodgepodge of topics, it is a carefully selected collection of features and confusing aspects of the language.

REFERENCES

Professional C++ code, including much of the code in this book, uses references extensively. It is helpful to step back and think about what exactly references are, and how they behave.

A *reference* in C++ is an *alias* for another variable. All modifications to the reference change the value of the variable to which it refers. You can think of references as implicit pointers that save you the trouble of taking the address of variables and dereferencing the pointer. Alternatively, you can think of references as just another name for the original variable. You can create stand-alone reference variables, use reference data members in classes, accept references as parameters to functions and methods, and return references from functions and methods.

Reference Variables

Reference variables must be initialized as soon as they are created, like this:

```
int x = 3;
int& xRef = x;
```

Subsequent to this assignment, `xRef` is another name for `x`. Any use of `xRef` uses the current value of `x`. Any assignment to `xRef` changes the value of `x`. For example, the following code sets `x` to 10 through `xRef`:

```
xRef = 10;
```

You cannot declare a reference variable outside of a class without initializing it:

```
int& emptyRef; // DOES NOT COMPILE!
```

WARNING *You must always initialize a reference when it is created. Usually, references are created when they are declared, but reference data members need to be initialized in the constructor initializer for the containing class.*

You cannot create a reference to an unnamed value, such as an integer literal, unless the reference is to a `const` value. In the following example, `unnamedRef1` will not compile because it is a non-`const` reference to a constant. That would mean you could change the value of the constant, 5, which doesn't make sense. `unnamedRef2` works because it's a `const` reference, so you cannot write "`unnamedRef2 = 7`".

```
int& unnamedRef1 = 5; // DOES NOT COMPILE
const int& unnamedRef2 = 5; // Works
```

Modifying References

A reference always refers to the same variable to which it is initialized; references cannot be changed once they are created. This rule leads to some confusing syntax. If you “assign” a variable to a reference when the reference is declared, the reference refers to that variable. However, if you assign a variable to a reference after that, the variable to which the reference refers is changed to the value of the variable being assigned. The reference is not updated to refer to that variable. Here is a code example:

```
int x = 3, y = 4;
int& xRef = x;
xRef = y; // Changes value of x to 4. Doesn't make xRef refer to y.
```

You might try to circumvent this restriction by taking the address of y when you assign it:

```
int x = 3, y = 4;
int& xRef = x;
xRef = &y; // DOES NOT COMPILE!
```

This code does not compile. The address of y is a pointer, but xRef is declared as a reference to an int, not a reference to a pointer.

Some programmers go even further in attempts to circumvent the intended semantics of references. What if you assign a reference to a reference? Won’t that make the first reference refer to the variable to which the second reference refers? You might be tempted to try this code:

```
int x = 3, z = 5;
int& xRef = x;
int& zRef = z;
zRef = xRef; // Assigns values, not references
```

The final line does not change zRef. Instead, it sets the value of z to 3, because xRef refers to x, which is 3.

WARNING *You cannot change the variable to which a reference refers after it is initialized; you can change only the value of that variable.*

References to Pointers and Pointers to References

You can create references to any type, including pointer types. Here is an example of a reference to a pointer to int:

```
int* intP;
int*& ptrRef = intP;
ptrRef = new int;
*ptrRef = 5;
```

The syntax is a little strange: You might not be accustomed to seeing * and & right next to each other. However, the semantics are straightforward: ptrRef is a reference to intP, which is a pointer

to int. Modifying `ptrRef` changes `intP`. References to pointers are rare, but can occasionally be useful, as discussed in the “Reference Parameters” section later in this chapter.

Note that taking the address of a reference gives the same result as taking the address of the variable to which the reference refers. For example:

```
int x = 3;
int& xRef = x;
int* xPtr = &xRef; // Address of a reference is pointer to value
*xPtr = 100;
```

This code sets `xPtr` to point to `x` by taking the address of a reference to `x`. Assigning 100 to `*xPtr` changes the value of `x` to 100. Writing a comparison “`xPtr == xRef`” will not compile because of a type mismatch; `xPtr` is a pointer to an int while `xRef` is a reference to an int. The comparisons “`xPtr == &xRef`” and “`xPtr == &x`” both compile without errors and are both `true`.

Finally, note that you cannot declare a reference to a reference, or a pointer to a reference.

Reference Data Members

As Chapter 8 explains, data members of classes can be references. A reference cannot exist without referring to some other variable. Thus, you must initialize reference data members in the constructor initializer, not in the body of the constructor. The following is a quick example:

```
class MyClass
{
public:
    MyClass(int& ref) : mRef(ref) {}
private:
    int& mRef;
};
```

Consult Chapter 8 for details.

Reference Parameters

C++ programmers do not often use stand-alone reference variables or reference data members. The most common use of references is for parameters to functions and methods. Recall that the default parameter-passing semantics are pass-by-value: Functions receive copies of their arguments. When those parameters are modified, the original arguments remain unchanged. References allow you to specify pass-by-reference semantics for arguments passed to the function. When you use reference parameters, the function receives references to the function arguments. If those references are modified, the changes are reflected in the original argument variables. For example, here is a simple swap function to swap the values of two ints:

```
void swap(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}
```

You can call it like this:

```
int x = 5, y = 6;
swap(x, y);
```

When the function `swap()` is called with the arguments `x` and `y`, the first parameter is initialized to refer to `x`, and the second parameter is initialized to refer to `y`. When `swap()` modifies `first` and `second`, `x` and `y` are actually changed.

Just as you can't initialize normal reference variables with constants, you can't pass constants as arguments to functions that employ pass-by-reference:

```
swap(3, 4); // DOES NOT COMPILE
```

NOTE *Using rvalue references, it is possible to pass constants as arguments to functions that employ pass-by-rvalue-reference. Rvalue references are discussed later in this chapter.*

References from Pointers

A common quandary arises when you have a pointer to something that you need to pass to a function or method that takes a reference. You can “convert” a pointer to a reference in this case by dereferencing the pointer. This action gives you the value to which the pointer points, which the compiler then uses to initialize the reference parameter. For example, you can call `swap()` like this:

```
int x = 5, y = 6;
int *xp = &x, *yp = &y;
swap(*xp, *yp);
```

Pass-by-Reference Versus Pass-by-Value

Pass-by-reference is required when you want to modify the parameter and see those changes reflected in the variable passed to the function or method. However, you should not limit your use of pass-by-reference to only those cases. Pass-by-reference avoids copying the arguments to the function, providing two additional benefits in some cases:

- 1. Efficiency:** Large objects and structs could take a long time to copy. Pass-by-reference passes only a pointer to the object or struct into the function.
- 2. Correctness:** Not all objects allow pass-by-value. Even those that do allow it might not support deep copying correctly. As Chapter 8 explains, objects with dynamically allocated memory must provide a custom copy constructor in order to support deep copying.

If you want to leverage these benefits, but do not want to allow the original objects to be modified, you can mark the parameters `const`, giving you pass-by-const-reference. This topic is covered in detail later in this chapter.

These benefits to pass-by-reference imply that you should use pass-by-value only for simple built-in types like `int` and `double` for which you don't need to modify the arguments. Use pass-by-reference in all other cases.

Reference Return Values

You can also return a reference from a function or method. The main reason to do so is efficiency. Instead of returning a whole object, return a reference to the object to avoid copying it unnecessarily. Of course, you can only use this technique if the object in question will continue to exist following the function termination.

WARNING *From a function or method, never return a reference to a variable that is locally scoped to that function or method, such as an automatically allocated variable on the stack that will be destroyed when the function ends.*

If the type you want to return from your function supports move semantics, discussed later in this chapter, then returning it by value is almost as efficient as returning a reference.

A second reason to return a reference is if you want to be able to assign to the return value directly as an *lvalue* (the left-hand side of an assignment statement). Several overloaded operators commonly return references. Chapter 8 shows some examples, and you can read about more applications of this technique in Chapter 14.

Deciding between References and Pointers

References in C++ could be considered redundant: almost everything you can do with references, you can accomplish with pointers. For example, you could write the previously shown `swap()` function like this:

```
void swap(int* first, int* second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}
```

However, this code is more cluttered than the version with references: References make your programs cleaner and easier to understand. References are also safer than pointers: It's impossible to have a null reference, and you don't explicitly dereference references, so you can't encounter any of the dereferencing errors associated with pointers. These arguments, saying that references are safer, are only valid in the absence of any pointers. For example, take the following function that accepts a reference to an `int`:

```
void refcall(int& t) { ++t; }
```

You could declare a pointer and initialize it to point to some random place in memory. Then you could dereference this pointer and pass it as the reference argument to `refcall()`, as in the following code. This code will compile but will crash on execution.

```
int* ptr = (int*)8;
refcall(*ptr);
```

Most of the time, you can use references instead of pointers. References to objects even support polymorphism in the same way as pointers to objects. The only case in which you need to use a pointer is when you need to change the location to which it points. Recall that you cannot change the variable to which references refer. For example, when you dynamically allocate memory, you need to store a pointer to the result in a pointer rather than a reference. A second use-case in which you need to use a pointer is for optional parameters. A pointer parameter can, for example, be defined as optional with a default value of `nullptr`, something that is not possible with a reference parameter.

Another way to distinguish between appropriate use of pointers and references in parameters and return types is to consider who *owns* the memory. If the code receiving the variable becomes the owner and thus becomes responsible for releasing the memory associated with an object, it must receive a pointer to the object, or better yet a smart pointer, which is the recommended way to transfer ownership. If the code receiving the variable should not free the memory, it should receive a reference.

NOTE *Use references instead of pointers unless you need to change to where the reference refers to.*

Consider a function that splits an array of `int`s into two arrays: one of even numbers and one of odd numbers. The function doesn't know how many numbers in the source array will be even or odd, so it should dynamically allocate the memory for the destination arrays after examining the source array. It should also return the sizes of the two new arrays. Altogether, there are four items to return: pointers to the two new arrays and the sizes of the two new arrays. Obviously, you must use pass-by-reference. The canonical C way to write the function looks like this:

```
void separateOddsAndEvens(const int arr[], int size, int** odds,
    int* numOdds, int** evens, int* numEvens)
{
    // Count the number of odds and evens
    *numOdds = *numEvens = 0;
    for (int i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            ++(*numOdds);
        } else {
            ++(*numEvens);
        }
    }
    // Allocate two new arrays of the appropriate size.
    *odds = new int[*numOdds];
    *evens = new int[*numEvens];
    // Copy the odds and evens to the new arrays
    int oddsPos = 0, evensPos = 0;
    for (int i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            (*odds)[oddsPos++] = arr[i];
        } else {
            (*evens)[evensPos++] = arr[i];
        }
    }
}
```

The final four parameters to the function are the “reference” parameters. In order to change the values to which they refer, `separateOddsAndEvens()` must dereference them, leading to some ugly syntax in the function body. Additionally, when you want to call `separateOddsAndEvens()`, you must pass the address of two pointers so that the function can change the actual pointers, and the address of two `ints` so that the function can change the actual `ints`:

```
int unSplit[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *oddNums, *evenNums;
int numOdds, numEvens;
separateOddsAndEvens(unSplit, 10, &oddNums, &numOdds, &evenNums, &numEvens);
```

If such syntax annoys you (which it should), you can write the same function by using references to obtain true pass-by-reference semantics:

```
void separateOddsAndEvens(const int arr[], int size, int*& odds,
                           int& numOdds, int*& evens, int& numEvens)
{
    numOdds = numEvens = 0;
    for (int i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            ++numOdds;
        } else {
            ++numEvens;
        }
    }
    odds = new int[numOdds];
    evens = new int[numEvens];
    int oddsPos = 0, evensPos = 0;
    for (int i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            odds[oddsPos++] = arr[i];
        } else {
            evens[evensPos++] = arr[i];
        }
    }
}
```

In this case, the `odds` and `evens` parameters are references to `int*`s. `separateOddsAndEvens()` can modify the `int*`s that are used as arguments to the function (through the reference), without any explicit dereferencing. The same logic applies to `numOdds` and `numEvens`, which are references to `ints`. With this version of the function, you no longer need to pass the addresses of the pointers or `ints`. The reference parameters handle it for you automatically:

```
int unSplit[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *oddNums, *evenNums;
int numOdds, numEvens;
separateOddsAndEvens(unSplit, 10, oddNums, numOdds, evenNums, numEvens);
```

It’s recommended to avoid dynamically allocated arrays as much as possible. For example, by using the STL `vector` container, the previous `separateOddsAndEvens()` can be rewritten to be more safe and elegant, because all memory allocation and deallocation happens automatically:

```
void separateOddsAndEvens(const vector<int>& arr,
                           vector<int>& odds, vector<int>& evens)
```

```

    {
        int numOdds = 0, numEvens = 0;
        for (auto& i : arr) {
            if (i % 2 == 1) {
                ++numOdds;
            } else {
                ++numEvens;
            }
        }
        odds.reserve(numOdds);
        evens.reserve(numEvens);
        for (auto& i : arr) {
            if (i % 2 == 1) {
                odds.push_back(i);
            } else {
                evens.push_back(i);
            }
        }
    }
}

```

This version can be used as follows:

```

vector<int> vecUnSplit = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
vector<int> odds, evens;
separateOddsAndEvens(vecUnSplit, odds, evens);

```

The STL vector container is discussed in detail in Chapter 16.

Rvalue References

In C++, an *lvalue* is something of which you can take an address; a named variable, for example. The name comes from the fact that they normally appear on the left-hand side of an assignment. An *rvalue* on the other hand is anything that is not an lvalue such as a constant value, or a temporary object or value. Typically an rvalue is on the right-hand side of an assignment operator.

An *rvalue reference* is a reference to an rvalue. In particular, it is a concept that is applied when the rvalue is a temporary object. The purpose of an rvalue reference is to make it possible for a particular function to be chosen when a temporary object is involved. The consequence of this is that certain operations that normally involve copying large values can be implemented by copying pointers to those values, knowing the temporary object will be destroyed.

A function can specify an rvalue reference parameter by using `&&` as part of the parameter specification; e.g., `type&& name`. Normally, a temporary object will be seen as a `const type&`, but when there is a function overload that uses an rvalue reference, a temporary object can be resolved to that overload. The following example demonstrates this. The code first defines two `incr()` functions, one accepting an lvalue reference and one accepting an rvalue reference.

```

// Increment value using lvalue reference parameter.
void incr(int& value)
{
    cout << "increment with lvalue reference" << endl;
}

```

```
        ++value;
    }
// Increment value using rvalue reference parameter.
void incr(int&& value)
{
    cout << "increment with rvalue reference" << endl;
    ++value;
}
```

You can call the `incr()` function with a named variable as argument. Because `a` is a named variable, the `incr()` function accepting an lvalue reference is called. After the call to `incr()`, the value of `a` will be 11.

```
int a = 10, b = 20;
incr(a);           // Will call incr(int& value)
```

You can also call the `incr()` function with an expression as argument. The `incr()` function accepting an lvalue reference cannot be used, because the expression `a + b` results in a temporary, which is not an lvalue. In this case the rvalue reference version is called. Since the argument is a temporary, the incremented value is lost after the call to `incr()`.

```
incr(a + b);      // Will call incr(int&& value)
```

A literal can also be used as argument to the `incr()` call. This will also trigger a call to the rvalue reference version because a literal cannot be an lvalue.

```
incr(3);          // Will call incr(int&& value)
```

If you remove the `incr()` function accepting an lvalue reference, calling `incr()` with a named variable like `incr(b)` will result in a compiler error because an rvalue reference parameter (`int&& value`) will never be bound to an lvalue (`b`). You can force the compiler to call the rvalue reference version of `incr()` by using `std::move()`, which converts an lvalue into an rvalue as follows. After the `incr()` call, the value of `b` will be 21.

```
incr(std::move(b)); // Will call incr(int&& value)
```

Rvalue references are not limited to parameters of functions. You can declare a variable of type rvalue reference, and assign to it, although this usage is uncommon. Consider the following code, which is illegal in C++:

```
int& i = 2;          // Invalid: reference to a constant
int a = 2, b = 3;
int& j = a + b;    // Invalid: reference to a temporary
```

Using rvalue references, the following is perfectly legal:

```
int&& i = 2;
int a = 2, b = 3;
int&& j = a + b;
```

Stand-alone rvalue references, as in the preceding example, are rarely used as such.

Move Semantics

Move semantics for objects requires a *move constructor* and a *move assignment operator*. These will be used by the compiler on places where the source object is a temporary object that will be destroyed after the copy or assignment. Both the move constructor and the move assignment operator copy/move the member variables from the source object to the new object and then reset the variables of the source object to null values. By doing this, they are actually *moving ownership* of the memory from one object to another object. They basically do a *shallow* copy of the member variables and switch ownership of allocated memory to prevent dangling pointers or memory leaks.

Move semantics is implemented by using rvalue references. To add move semantics to a class, a move constructor and a move assignment operator need to be implemented. Move constructors and move assignment operators should be marked with the `noexcept` qualifier to tell the compiler that they don't throw any exceptions. This is particularly important for compatibility with the standard library, as fully compliant implementations of the standard library will only move stored objects if, having move semantics implemented, they also guarantee not to throw. Following is the `Spreadsheet` class definition from Chapter 8, with a move constructor and move assignment operator added:

```
class Spreadsheet
{
public:
    Spreadsheet(Spreadsheet&& src) noexcept; // Move constructor
    Spreadsheet& operator=(Spreadsheet&& rhs) noexcept; // Move assignment
    // Remaining code omitted for brevity
};
```

The implementation is as follows. Note that a helper method is introduced called `freeMemory()`, which deallocates the `mCells` array (not shown). This helper method is called from the destructor, the normal assignment operator, and the move assignment operator. Similarly, a helper method can be added to move the data from a source object to a destination object, which can then be used from the move constructor and the move assignment operator.

```
// Move constructor
Spreadsheet::Spreadsheet(Spreadsheet&& src) noexcept
{
    // Shallow copy of data
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    mCells = src.mCells;
    // Reset the source object, because ownership has been moved!
    src.mWidth = 0;
    src.mHeight = 0;
    src.mCells = nullptr;
}
// Move assignment operator
Spreadsheet& Spreadsheet::operator=(Spreadsheet&& rhs) noexcept
{
    // check for self-assignment
    if (this == &rhs) {
        return *this;
    }
    // free the old memory
    freeMemory();
```

```
// Shallow copy of data
mWidth = rhs.mWidth;
mHeight = rhs.mHeight;
mCells = rhs.mCells;
// Reset the source object, because ownership has been moved!
rhs.mWidth = 0;
rhs.mHeight = 0;
rhs.mCells = nullptr;
return *this;
}
```

Both the move constructor and the move assignment operator are moving ownership of the memory for `mCells` from the source object to the new object. They reset the `mCells` pointer of the source object to a null pointer to prevent the destructor of the source object to deallocate that memory because now the new object is the owner of that memory.

The preceding move constructor and move assignment operator can be tested with the following code:

```
Spreadsheet CreateObject()
{
    return Spreadsheet(3, 2);
}
int main()
{
    vector<Spreadsheet> vec;
    for (int i = 0; i < 2; ++i) {
        cout << "Iteration " << i << endl;
        vec.push_back(Spreadsheet(100, 100));
        cout << endl;
    }
    Spreadsheet s(2,3);
    s = CreateObject();
    Spreadsheet s2(5,6);
    s2 = s;
    return 0;
}
```

Chapter 1 introduces the `vector`. A `vector` grows dynamically in size to accommodate new objects. This is done by allocating a bigger chunk of memory and then copying or moving the objects from the old `vector` to the new and bigger `vector`. If the compiler finds a move constructor, the objects will be moved instead of copied. By moving them there is no need for any deep copying, making it much more efficient.

When you add output statements to all constructors and assignment operators of the `Spreadsheet` class, the output of the preceding test program will be as follows. This output and the following discussion is based on Microsoft Visual C++ 2013. The C++ standard does not specify the initial capacity of a `vector` nor its growth strategy, so the output can be different on different compilers.

Iteration 0	
Normal constructor	(1)
Move constructor	(2)

Iteration 1	
Normal constructor	(3)
Move constructor	(4)
Move constructor	(5)
Normal constructor	(6)
Normal constructor	(7)
Move assignment operator	(8)
Normal constructor	(9)
Assignment operator	(10)

On the first iteration of the loop, the `vector` is still empty. Take the following line of code from the loop:

```
vec.push_back(Spreadsheet(100, 100));
```

With this line, a new `Spreadsheet` object is created invoking the normal constructor (1). The `vector` resizes itself to make space for the new object being pushed in. The created `Spreadsheet` object is then moved into the `vector`, invoking the move constructor (2).

On the second iteration of the loop, a second `Spreadsheet` object is created with the normal constructor (3). At this point, the `vector` can hold one element, so it's again resized to make space for a second object. By resizing the `vector`, the previously added elements need to be moved from the old `vector` to the new and bigger `vector`, so this will trigger a call to the move constructor for each previously added element (4). Then, the new `Spreadsheet` object is moved into the `vector` with its move constructor (5).

Next, a `Spreadsheet` object `s` is created using the normal constructor (6). The `CreateObject()` function creates a temporary `Spreadsheet` object with its normal constructor (7), which is then returned from the function and move-assigned to the variable `s` (8). Because the temporary object will cease to exist after the assignment, the compiler will invoke the move assignment operator instead of the normal copy assignment operator. On the other hand, the assignment `s2 = s` will invoke the copy assignment operator (10) because the right-hand side object is not a temporary object, but a named object.

If the `Spreadsheet` class did not include a move constructor and move assignment operator, the above output would look as follows:

```
Iteration 0
Normal constructor
Copy constructor

Iteration 1
Normal constructor
Copy constructor
Copy constructor

Normal constructor
Normal constructor
Assignment operator
Normal constructor
Assignment operator
```

As you can see, copy constructors are called instead of move constructors and copy assignment operators are called instead of move assignment operators. In the previous example, the `Spreadsheet` objects in the loop have 10,000 (100 x 100) elements. The implementation of the `Spreadsheet` move constructor and move assignment operator don't require any memory allocation, while the copy constructor and copy assignment operator require 101 allocations. So, using move semantics can increase performance a lot in certain situations.

Move constructors and move assignment operators can also be explicitly deleted or defaulted, just like normal constructors and normal copy assignment operators, as explained in Chapter 7.

As another example where move semantics increases performance, take a `swap()` function that swaps two objects. The following `swapCopy()` implementation does not use move semantics:

```
void swapCopy(T& a, T& b)
{
    T temp(a);
    a = b;
    b = temp;
}
```

This implementation first copies `a` to `temp`, then copies `b` to `a`, and then copies `temp` to `b`. If type `T` is expensive to copy, this swap implementation will hurt performance. With move semantics the `swap()` function can avoid all copying:

```
void swapMove(T& a, T& b)
{
    T temp(std::move(a));
    a = std::move(b);
    b = std::move(temp);
}
```

Obviously, move semantics is useful only when you know that the source object will be destroyed.

KEYWORD CONFUSION

Two keywords in C++ appear to cause more confusion than any others: `const` and `static`. Both of these keywords have several different meanings, and each of their uses presents subtleties that are important to understand.

The `const` Keyword

The keyword `const` is short for “constant” and specifies that something remains unchanged. The compiler will enforce this requirement by marking any attempt to change it as an error. Furthermore, when optimizations are enabled, the compiler can take advantage of this knowledge to produce better code. The keyword has two related roles. It can mark variables or parameters, and it can mark methods. This section provides a definitive discussion of these two meanings.

const Variables and Parameters

You can use `const` to “protect” variables by specifying that they cannot be modified. One important use is as a replacement for `#define` to define constants. This use of `const` is its most straightforward application. For example, you could declare the constant `PI` like this:

```
const double PI = 3.141592653589793238462;
```

You can mark any variable `const`, including global variables and class data members.

You can also use `const` to specify that parameters to functions or methods should remain unchanged. For example, the following function accepts a `const` parameter. In the body of the function, you cannot modify the `param` integer. If you do try to modify it, the compiler will generate an error.

```
void func(const int param)
{
    // Not allowed to change param...
}
```

The following subsections discuss two special kinds of `const` variables or parameters in more detail: `const` pointers and `const` references.

const Pointers

When a variable contains one or more levels of indirection via a pointer, applying `const` becomes trickier. Consider the following lines of code:

```
int* ip;
ip = new int[10];
ip[4] = 5;
```

Suppose that you decide to apply `const` to `ip`. Set aside your doubts about the usefulness of doing so for a moment, and consider what it means. Do you want to prevent the `ip` variable itself from being changed, or do you want to prevent the values to which it points from being changed? That is, do you want to prevent the second line or the third line in the previous example?

In order to prevent the pointed-to values from being modified (as in the third line), you can add the keyword `const` to the declaration of `ip` like this:

```
const int* ip;
ip = new int[10];
ip[4] = 5; // DOES NOT COMPILE!
```

Now you cannot change the values to which `ip` points.

An alternative but semantically equivalent way to write this is as follows:

```
int const* ip;
ip = new int[10];
ip[4] = 5; // DOES NOT COMPILE!
```

Putting the `const` before or after the `int` makes no difference in its functionality.

If you want instead to mark `ip` itself `const` (not the values to which it points), you need to write this:

```
int* const ip = nullptr;
ip = new int[10]; // DOES NOT COMPILE!
ip[4] = 5;        // Error: dereferencing a null pointer
```

Now that `ip` itself cannot be changed, the compiler requires you to initialize it when you declare it, either with `nullptr` as in the preceding code or with newly allocated memory as follows:

```
int* const ip = new int[10];
ip[4] = 5;
```

You can also mark both the pointer and the values to which it points `const` like this:

```
int const* const ip = nullptr;
```

An alternative but equivalent syntax is the following:

```
const int* const ip = nullptr;
```

Although this syntax might seem confusing, there is actually a very simple rule: the `const` keyword applies to whatever is directly to its left. Consider this line again:

```
int const* const ip = nullptr;
```

From left to right, the first `const` is directly to the right of the word `int`. Thus, it applies to the `int` to which `ip` points. Therefore, it specifies that you cannot change the values to which `ip` points. The second `const` is directly to the right of the `*`. Thus, it applies to the pointer to the `int`, which is the `ip` variable. Therefore, it specifies that you cannot change `ip` (the pointer itself).

The reason this rule becomes confusing is an exception: The first `const` can go before the variable like this:

```
const int* const ip = nullptr;
```

This “exceptional” syntax is used much more commonly than the other syntax.

You can extend this rule to any number of levels of indirection. For example:

```
const int * const * const * const ip = nullptr;
```

NOTE *Another easy-to-remember rule to figure out complicated variable declarations: read from right to left. Take for example “`int* const ip`.” Reading this from right to left gives us “`ip` is a `const` pointer to an `int`.” On the other hand, “`int const* ip`” will read as “`ip` is a pointer to a `const int`.”*

const References

`const` applied to references is usually simpler than `const` applied to pointers for two reasons. First, references are `const` by default, in that you can't change to what they refer. So, there is no need to mark them `const` explicitly. Second, you can't create a reference to a reference, so there is usually only one level of indirection with references. The only way to get multiple levels of indirection is to create a reference to a pointer.

Thus, when C++ programmers refer to a “const reference,” they mean something like this:

```
int z;
const int& zRef = z;
zRef = 4; // DOES NOT COMPILE
```

By applying `const` to the `int`, you prevent assignment to `zRef`, as shown. Remember that `const int& zRef` is equivalent to `int const& zRef`. Note, however, that marking `zRef const` has no effect on `z`. You can still modify the value of `z` by changing it directly instead of through the reference.

`const` references are used most commonly as parameters, where they are quite useful. If you want to pass something by reference for efficiency, but don’t want it to be modifiable, make it a `const` reference. For example:

```
void doSomething(const BigClass& arg)
{
    // Implementation here
}
```

WARNING *Your default choice for passing objects as parameters should be `const` reference. You should only omit the `const` if you explicitly need to change the object.*

const Methods

Chapter 8 explains that you can mark a class method `const`, which prevents the method from modifying any non-`mutable` data members of the class. Consult Chapter 8 for an example.

The `constexpr` Keyword

C++ always had the notion of constant expressions and in some circumstances constant expressions are required. For example, when defining an array, the size of the array needs to be a constant expression. Because of this restriction, the following piece of code is not valid in C++.

```
const int getArraySize() { return 32; }
int main()
{
    int myArray[getArraySize()];    // Invalid in C++
    return 0;
}
```

Using the `constexpr` keyword, the `getArraySize()` function can be redefined to make it a constant expression. Constant expressions are evaluated at compile time.

```
constexpr int getArraySize() { return 32; }
int main()
{
    int myArray[getArraySize()];    // OK
    return 0;
}
```

You can even do something like this:

```
int myArray[getArraySize() + 1]; // OK
```

Declaring a function as `constexpr` imposes quite a lot of restrictions on what the function can do because the compiler has to be able to evaluate the function at compile time, and the function is not allowed to have any side effects. Here are a couple of restrictions:

- The function body shall be a single return statement that does not contain a `goto` statement or a try catch block, and does not throw any exceptions. It is allowed to call other `constexpr` functions.
- The return type of the function shall be a literal type. It cannot be `void`.
- If the `constexpr` function is a member of a class, the function cannot be `virtual`.
- All the function arguments shall be literal types.
- A `constexpr` function cannot be called until it's defined in the translation unit because the compiler needs to know the complete definition.
- `dynamic_cast` is not allowed.
- `new` and `delete` are not allowed.

By defining a `constexpr` constructor you can create constant expression variables of user-defined types. A `constexpr` constructor should satisfy the following requirements.

- All the constructor arguments should be literal types.
- The constructor body cannot be a function-try-block.
- The constructor body should satisfy the same requirements as the body of a `constexpr` function.
- All data members should be initialized with constant expressions.

For example, the following `Rect` class defines a `constexpr` constructor satisfying the previous requirements and also defines a `constexpr` `getArea()` method that is performing some calculation.

```
class Rect
{
public:
    constexpr Rect(int width, int height)
        : mWidth(width), mHeight(height) {}
    constexpr int getArea() const { return mWidth * mHeight; }
private:
    int mWidth, mHeight;
};
```

Using this class to declare a `constexpr` object is straightforward:

```
constexpr Rect r(8, 2);
int myArray[r.getArea()]; // OK
```

The static Keyword

There are several uses of the keyword `static` in C++, all seemingly unrelated. Part of the motivation for “overloading” the keyword was attempting to avoid having to introduce new keywords into the language.

static Data Members and Methods

You can declare `static` data members and methods of classes. `static` data members, unlike non-`static` data members, are not part of each object. Instead, there is only one copy of the data member, which exists outside any objects of that class.

`static` methods are similarly at the class level instead of the object level. A `static` method does not execute in the context of a specific object.

Chapter 8 provides examples of both `static` data members and methods.

static Linkage

Before covering the use of the `static` keyword for linkage, you need to understand the concept of *linkage* in C++. C++ source files are each compiled independently, and the resulting object files are linked together. Each name in a C++ source file, including functions and global variables, has a linkage that is either *internal* or *external*. External linkage means that the name is available from other source files. Internal linkage (also called *static linkage*) means that it is not. By default, functions and global variables have external linkage. However, you can specify internal (or static) linkage by prefixing the declaration with the keyword `static`. For example, suppose you have two source files: `FirstFile.cpp` and `AnotherFile.cpp`. Here is `FirstFile.cpp`:

```
void f();
int main()
{
    f();
    return 0;
}
```

Note that this file provides a prototype for `f()`, but doesn’t show the definition. Here is `AnotherFile.cpp`:

```
#include <iostream>
using namespace std;
void f();
void f()
{
    cout << "f\n";
}
```

This file provides both a prototype and a definition for `f()`. Note that it is legal to write prototypes for the same function in two different files. That’s precisely what the preprocessor does for you if you put the prototype in a header file that you `#include` in each of the source files. The reason to use header files is that it’s easier to maintain (and keep synchronized) one copy of the prototype. However, for this example I don’t use a header file.

Each of these source files compiles without error, and the program links fine: because `f()` has external linkage, `main()` can call it from a different file.

However, suppose you apply `static` to the `f()` prototype in `AnotherFile.cpp`. Note that you don't need to repeat the `static` keyword in front of the definition of `f()`. As long as it precedes the first instance of the function name, there is no need to repeat it:

```
#include <iostream>
using namespace std;
static void f();
void f()
{
    cout << "f\n";
}
```

Now each of the source files compiles without error, but the linker step fails because `f()` has internal (`static`) linkage, making it unavailable from `FirstFile.cpp`. Some compilers issue a warning when `static` methods are defined but not used in that source file (implying that they shouldn't be `static`, because they're probably used elsewhere).

An alternative to using `static` for internal linkage is to employ *anonymous namespaces*. Instead of marking a variable or function `static`, wrap it in an unnamed namespace like this:

```
#include <iostream>
using namespace std;
namespace {
    void f();
    void f()
    {
        cout << "f\n";
    }
}
```

Entities in an anonymous namespace can be accessed anywhere following their declaration in the same source file, but cannot be accessed from other source files. These semantics are the same as those obtained with the `static` keyword.

The `extern` Keyword

A related keyword, `extern`, seems like it should be the opposite of `static`, specifying external linkage for the names it precedes. It can be used that way in certain cases. For example, `consts` and `typedefs` have internal linkage by default. You can use `extern` to give them external linkage. However, `extern` has some complications. When you specify a name as `extern`, the compiler treats it as a declaration, not a definition. For variables, this means the compiler doesn't allocate space for the variable. You must provide a separate definition line for the variable without the `extern` keyword. For example:

```
extern int x;
int x = 3;
```

Alternatively, you can initialize `x` in the `extern` line, which then serves as the declaration and definition:

```
extern int x = 3;
```

The `extern` in this case is not very useful, because `x` has external linkage by default anyway. The real use of `extern` is when you want to use `x` from another source file, `FirstFile.cpp`:

```
#include <iostream>
using namespace std;
extern int x;
int main()
{
    cout << x << endl;
}
```

Here `FirstFile.cpp` uses an `extern` declaration so that it can use `x`. The compiler needs a declaration of `x` in order to use it in `main()`. If you declared `x` without the `extern` keyword, the compiler would think it's a definition and would allocate space for `x`, causing the linkage step to fail (because there are now two `x` variables in the global scope). With `extern`, you can make variables globally accessible from multiple source files.

However, it is not recommended to use global variables at all. They are confusing and error-prone, especially in large programs. For similar functionality, you could use `static` class data members and methods.

static Variables in Functions

The final use of the `static` keyword in C++ is to create local variables that retain their values between exits and entrances to their scope. A `static` variable inside a function is like a global variable that is only accessible from that function. One common use of `static` variables is to “remember” whether a particular initialization has been performed for a certain function. For example, code that employs this technique might look something like this:

```
void performTask()
{
    static bool initialized = false;
    if (!initialized) {
        cout << "initializing\n";
        // Perform initialization.
        initialized = true;
    }
    // Perform the desired task.
}
```

However, `static` variables are confusing, and there are usually better ways to structure your code so that you can avoid them. In this case, you might want to write a class in which the constructor performs the required initialization.

NOTE *Avoid using stand-alone static variables. Maintain state within an object instead.*

NOTE *The implementation of `performTask()` is not thread-safe; it contains a race condition. In a multithreaded environment, you need to use atomics or other mechanisms for synchronization of multiple threads. Multithreading is discussed in detail in Chapter 23.*

Order of Initialization of Nonlocal Variables

Before leaving the topic of `static` data members and global variables, consider the order of initialization of these variables. All global variables and `static` class data members in a program are initialized before `main()` begins. The variables in a given source file are initialized in the order they appear in the source file. For example, in the following file `Demo.cpp` `x` is guaranteed to be initialized before `y`:

```
class Demo
{
public:
    static int x;
};

int Demo::x = 3;
int y = 4;
```

However, C++ provides no specifications or guarantees about the initialization ordering of nonlocal variables in different source files. If you have a global variable `x` in one source file and a global variable `y` in another, you have no way of knowing which will be initialized first. Normally, this lack of specification isn't cause for concern. However, it can be problematic if one global or `static` variable depends on another. Recall that initialization of objects implies running their constructors. The constructor of one global object might access another global object, assuming that it is already constructed. If these two global objects are declared in two different source files, you cannot count on one being constructed before the other, and you cannot control the order of initialization. This order may not be the same for different compilers or even different versions of the same compiler, and the order might even change by simply adding another file to your project.

WARNING *Initialization order of nonlocal variables in different source files is undefined.*

Order of Destruction of Nonlocal Variables

Nonlocal variables are destroyed in the reverse order they were initialized. Nonlocal variables in different source files are initialized in an undefined order, which means that the order of destruction is also undefined.

TYPES AND CASTS

The basic types in C++ are reviewed in Chapter 1, while Chapter 7 shows you how to write your own types with classes. This section explores some of the trickier aspects of types: `typedefs`, `typedefs` for function pointers, type aliases, and casts.

typedefs

A `typedef` provides a new name for an existing type declaration. You can think of a `typedef` as syntax for introducing a synonym for an existing type declaration without creating a new type. The following gives the new name `IntPtr` to the `int*` type declaration:

```
typedef int* IntPtr;
```

You can use the new type name and the definition it aliases interchangeably. For example, the following two lines are valid:

```
int* p1;
IntPtr p2;
```

Variables created with the new type name are completely compatible with those created with the original type declaration. So it is perfectly valid, given the above definitions, to write the following, because they are not just “compatible” types, they are the same type:

```
p1 = p2;
p2 = p1;
```

The most common use of `typedefs` is to provide manageable names when the real type declarations become too unwieldy. This situation commonly arises with templates. For example, Chapter 1 introduces the `std::vector` from the STL. To declare a vector of strings, you need to declare it as `std::vector<std::string>`. It’s a templated class, and thus requires you to specify the template parameters anytime you want to refer to the type of this vector. Templates are discussed in detail in Chapter 11. For declaring variables, specifying function parameters, and so on, you would have to write `std::vector<std::string>`:

```
void processVector(const std::vector<std::string>& vec) { /* omitted */ }
int main()
{
    std::vector<std::string> myVector;
    return 0;
}
```

With a `typedef`, you can create a shorter, more meaningful name:

```
typedef std::vector<std::string> StringVector;
void processVector(const StringVector& vec) { /* omitted */ }
int main()
{
    StringVector myVector;
    return 0;
}
```

typedefs can include the scope qualifiers. The preceding example shows this by including the scope `std` for `StringVector`.

The STL uses `typedefs` extensively to provide shorter names for types. For example, `string` is actually a `typedef` that looks like this:

```
typedef basic_string<char, char_traits<char>, allocator<char>> string;
```

typedefs for Function Pointers

The most convoluted use of `typedefs` is when defining function pointers. While function pointers in C++ are uncommon (being replaced by `virtual` methods), there are needs to obtain function pointers in certain cases. Perhaps the most common example of this is when obtaining a pointer to a function in a dynamic link library. The following example obtains a pointer to a function in a Microsoft Windows Dynamic Link Library (DLL). Details of Windows DLLs are outside the scope of this book on platform-independent C++, but it is so important to Windows programmers that it is worth discussing, and it is a good example to explain the details of function pointers in general.

Consider a DLL that has a function called `MyFunc()`. You would like to load this library only if you need to call `MyFunc()`. This at run-time loading of the library is done with the Windows `LoadLibrary()` kernel call:

```
HMODULE lib = ::LoadLibrary(_T("library name"));
```

The result of this call is what is called a “library handle” and will be `NULL` if there is an error. Before you can load the function from the library, you need to know the prototype for the function. Suppose the following is the prototype for `MyFunc()`, which returns an integer and accepts three parameters: a Boolean, an integer, and a C-style string.

```
int __stdcall MyFunc(bool b, int n, const char* p);
```

The `__stdcall` is a Microsoft-specific directive to specify how parameters are passed to the function and how they are cleaned up.

You can now use a `typedef` to define a short name (`MyFuncProc`) for a pointer to a function with the preceding prototype.

```
typedef int (__stdcall *MyFuncProc)(bool b, int n, const char* p);
```

Note that the `typedef` name `MyFuncProc` is embedded in the middle of the syntax. It is clear from this example that these kinds of `typedefs` are rather convoluted. The next section shows you a cleaner solution to this problem called type aliases.

Having successfully loaded the library and defined a short name for the function pointer, you can get a pointer to the function in the library as follows:

```
MyFuncProc MyProc = ::GetProcAddress(lib, "MyFunc");
```

If this fails, `MyProc` will be `NULL`. If it succeeds, you can call the loaded function:

```
MyProc(true, 3, "Hello world");
```

A C programmer might think that you need to dereference the function pointer before calling it as follows:

```
(*MyProc)(true, 3, "Hello world");
```

This was true decades ago, but now, every C and C++ compiler is smart enough to know how to automatically dereference the function pointer before calling it.

Type Aliases

Type aliases are easier to understand than `typedefs` in certain situations. For example, take the following `typedef`:

```
typedef int MyInt;
```

This can be written using a type alias as follows:

```
using MyInt = int;
```

This type alias feature is especially useful in cases where the `typedef` becomes complicated, which is the case with `typedefs` for function pointers as seen in the previous section. For example, the following `typedef` defines a type for a pointer to a function, which returns an integer and accepts a `char` and a `double` as parameters:

```
typedef int (*FuncType)(char, double);
```

This `typedef` is a bit convoluted because the name `FuncType` is somewhere in the middle of it. Using a type alias, this can be written as follows:

```
using FuncType = int (*)(char, double);
```

By reading through this section, you might think that type aliases are nothing more than easier-to-read `typedefs`, but there is more. The problem with `typedefs` becomes apparent when you want to use them with templates, but that is covered in Chapter 11 because it requires more details about templates.

Casts

C++ provides four specific casts: `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`.

The old C-style casts with `()` still work in C++, and are still used extensively in existing code bases. C-style casts cover all four C++ casts, so they are more error-prone because it's not always obvious what you are trying to achieve, and you might end up with unexpected results. I strongly recommend you only use the C++ style casts in new code because they are safer and stand out better syntactically in your code.

This section describes the purposes of each C++ cast and specifies when you would use each of them.

const_cast

The `const_cast` is the most straightforward. You can use it to cast away `const`-ness of a variable. It is the only cast of the four that is allowed to cast away `const`-ness. Theoretically, of course, there should be no need for a `const` cast. If a variable is `const`, it should stay `const`. In practice, however, you sometimes find yourself in a situation where a function is specified to take a `const` variable, which it must then pass to a function that takes a non-`const` variable. The “correct” solution would be to make `const` consistent in the program, but that is not always an option, especially if you are using third-party libraries. Thus, you sometimes need to cast away the `const`-ness of a variable, but you should only do this when you are sure the function you are calling will not modify the object, otherwise there is no other option than to restructure your program. Here is an example:

```
extern void ThirdPartyLibraryMethod(char* str);
void f(const char* str)
{
    ThirdPartyLibraryMethod(const_cast<char*>(str));
}
```

static_cast

You can use the `static_cast` to perform explicit conversions that are supported directly by the language. For example, if you write an arithmetic expression in which you need to convert an `int` to a `double` in order to avoid integer division, use a `static_cast`. In this example, it’s enough to only `static_cast i`, because that makes one of the two operands a `double`, making sure C++ performs floating point division.

```
int i = 3;
int j = 4;
double result = static_cast<double>(i) / j;
```

You can also use `static_cast` to perform explicit conversions that are allowed because of user-defined constructors or conversion routines. For example, if class `A` has a constructor that takes an object of class `B`, you can convert a `B` object to an `A` object with a `static_cast`. In most situations where you want this behavior, however, the compiler will perform the conversion automatically.

Another use for the `static_cast` is to perform downcasts in an inheritance hierarchy. For example:

```
class Base
{
public:
    Base() {}
    virtual ~Base() {}
};
class Derived : public Base
{
public:
    Derived() {}
    virtual ~Derived() {}
};
int main()
{
    Base* b;
    Derived* d = new Derived();
    b = d; // Don't need a cast to go up the inheritance hierarchy
```

```

d = static_cast<Derived*>(b); // Need a cast to go down the hierarchy
Base base;
Derived derived;
Base& br = derived;
Derived& dr = static_cast<Derived&>(br);
return 0;
}

```

These casts work with both pointers and references. They do not work with objects themselves.

Note that these casts with `static_cast` do not perform run-time type checking. They allow you to convert any `Base` pointer to a `Derived` pointer or `Base` reference to a `Derived` reference, even if the `base` really isn't a `Derived` at run time. For example, the following code will compile and execute, but using the pointer `d` can result in potentially catastrophic failure, including memory overwrites outside the bounds of the object.

```

Base* b = new Base();
Derived* d = static_cast<Derived*>(b);

```

To perform the cast safely with run-time type checking, use the `dynamic_cast` explained in a following section.

`static_casts` are not all-powerful. You can't `static_cast` pointers of one type to pointers of another unrelated type. You can't `static_cast` directly objects of one type to objects of another type. You can't `static_cast` a `const` type to a non-`const` type. You can't `static_cast` pointers to `ints`. Basically, you can't do anything that doesn't make sense according to the type rules of C++.

reinterpret_cast

The `reinterpret_cast` is a bit more powerful, and concomitantly less safe, than the `static_cast`. You can use it to perform some casts that are not technically allowed by C++ type rules, but which might make sense to the programmer in some circumstances. For example, you can cast a reference to one type to a reference to another type, even if the types are unrelated. Similarly, you can cast a pointer type to any other pointer type, even if they are unrelated by an inheritance hierarchy. This is commonly used to cast a pointer to a `void*` and back. A `void*` pointer is just a pointer to some location in memory. No type information is associated with a `void*` pointer. Here are some examples:

```

class X {};
class Y {};
int main()
{
    X x;
    Y y;
    X* xp = &x;
    Y* yp = &y;
    // Need reinterpret cast for pointer conversion from unrelated classes
    // static_cast doesn't work.
    xp = reinterpret_cast<X*>(yp);
    // No cast required for conversion from pointer to void*

```

```

void* p = xp;
// Need reinterpret cast for pointer conversion from void*
xp = reinterpret_cast<X*>(p);
// Need reinterpret cast for reference conversion from unrelated classes
// static_cast doesn't work.
X& xr = x;
Y& yr = reinterpret_cast<Y&>(x);
return 0;
}

```

You should be very careful with `reinterpret_cast` because it allows you to do conversions without performing any type checking.

WARNING *In theory, you could also use `reinterpret_cast` to cast pointers to ints and ints to pointers, but this is considered erroneous programming, because on many platforms (especially 64-bit platforms) pointers and ints are of different sizes. For example, on a 64-bit platform, pointers are 64 bit, but integers could be 32 bit. Casting a 64-bit pointer to a 32-bit integer will result in losing 32 critical bits!*

dynamic_cast

The `dynamic_cast` provides a run-time check on casts within an inheritance hierarchy. You can use it to cast pointers or references. `dynamic_cast` checks the run-time type information of the underlying object at run time. If the cast doesn't make sense, `dynamic_cast` returns a null pointer (for the pointer version) or throws an `std::bad_cast` exception (for the reference version).

Note that the run-time type information is stored in the vtable of the object. Therefore, in order to use `dynamic_cast`, your classes must have at least one virtual method. If your classes don't have a vtable, trying to use `dynamic_cast` will result in a compiler error, which can be a bit obscure. Microsoft VC++ for example will give the error:

```
error C2683: 'dynamic_cast' : 'MyClass' is not a polymorphic type.
```

Suppose you have the following class hierarchy:

```

class Base
{
public:
    Base() {}
    virtual ~Base() {}
};

class Derived : public Base
{
public:
    Derived() {}
    virtual ~Derived() {}
};

```

The following example shows a correct use of `dynamic_cast`.

```
Base* b;
Derived* d = new Derived();
b = d;
d = dynamic_cast<Derived*>(b);
```

The following `dynamic_cast` on a reference will cause an exception to be thrown.

```
Base base;
Derived derived;
Base& br = base;
try {
    Derived& dr = dynamic_cast<Derived&>(br);
} catch (const bad_cast&) {
    cout << "Bad cast!\n";
}
```

Note that you can perform the same casts down the inheritance hierarchy with a `static_cast` or `reinterpret_cast`. The difference with `dynamic_cast` is that it performs run-time (dynamic) type checking, while `static_cast` and `reinterpret_cast` will perform the casting even if they are erroneous.

Summary of Casts

The following table summarizes the casts you should use for different situations.

SITUATION	CAST
Remove <code>const</code> -ness	<code>const_cast</code>
Explicit cast supported by language (e.g., <code>int</code> to <code>double</code> , <code>int</code> to <code>bool</code>)	<code>static_cast</code>
Explicit cast supported by user-defined constructors or conversions	<code>static_cast</code>
Object of one class to object of another (unrelated) class	Can't be done
Pointer-to-object of one class to pointer-to-object of another class in the same inheritance hierarchy	<code>dynamic_cast</code> recommended, or <code>static_cast</code>
Reference-to-object of one class to reference-to-object of another class in the same inheritance hierarchy	<code>dynamic_cast</code> recommended, or <code>static_cast</code>
Pointer-to-type to unrelated pointer-to-type	<code>reinterpret_cast</code>
Reference-to-type to unrelated reference-to-type	<code>reinterpret_cast</code>
Pointer-to-function to pointer-to-function	<code>reinterpret_cast</code>

SCOPE RESOLUTION

As a C++ programmer, you need to familiarize yourself with the concept of *scope*. Every name in your program, including variable, function, and class names, is in a certain scope. You create scopes with namespaces, function definitions, blocks delimited by curly braces, and class definitions. When you try to access a variable, function, or class, the name is first looked up in the nearest enclosing scope, then the next scope, and so forth, up to the *global scope*. Any name not in a namespace, function, block delimited by curly braces, or class is assumed to be in the global scope. If it is not found in the global scope, at that point the compiler generates an undefined symbol error.

Sometimes names in scopes hide identical names in other scopes. Other times, the scope you want is not part of the default scope resolution from that particular line in the program. If you don't want the default scope resolution for a name, you can qualify the name with a specific scope using the scope resolution operator `::`. For example, to access a `static` method of a class, one way is to prefix the method name with the name of the class (its scope) and the scope resolution operator. A second way is to access the `static` method through an object of that class. The following example demonstrates these options. The example defines a class `Demo` with a `static` `get()` method, a `get()` function that is globally scoped, and a `get()` function that is in the `NS` namespace.

```
class Demo
{
public:
    static int get() { return 5; }
};

int get() { return 10; }

namespace NS
{
    int get() { return 20; }
}
```

The global scope is unnamed, but you can access it specifically by using the scope resolution operator by itself (with no name prefix). The different `get()` functions can be called as follows. In this example, the code itself is in the `main()` function, which is always in the global scope:

```
int main()
{
    auto pd = std::make_unique<Demo>();
    Demo d;
    std::cout << pd->get() << std::endl;      // prints 5
    std::cout << d.get() << std::endl;          // prints 5
    std::cout << NS::get() << std::endl;        // prints 20
    std::cout << Demo::get() << std::endl;       // prints 5
    std::cout << ::get() << std::endl;          // prints 10
    std::cout << get() << std::endl;            // prints 10
    return 0;
}
```

Note that if the namespace called `NS` is given as an unnamed namespace, then the following line will give an error about ambiguous name resolution, because you would have a `get()` defined in the global scope and another `get()` defined in the unnamed namespace.

```
std::cout << get() << std::endl;
```

The same error occurs if you add the following `using` clause right before the `main()` function:

```
using namespace NS;
```

C++11 / C++14

The C++11 and C++14 standards add a lot of new functionality to C++. This section describes new features of which detailed descriptions do not immediately fit elsewhere in this book.

Uniform Initialization

Before C++11, initialization of types was not always uniform. For example, take the following definition of a circle, once as a structure, once as a class:

```
struct CircleStruct
{
    int x, y;
    double radius;
};

class CircleClass
{
public:
    CircleClass(int x, int y, double radius)
        : mX(x), mY(y), mRadius(radius) {}
private:
    int mX, mY;
    double mRadius;
};
```

In pre-C++11, initialization of a variable of type `CircleStruct` and a variable of type `CircleClass` looks different:

```
CircleStruct myCircle1 = {10, 10, 2.5};
CircleClass myCircle2(10, 10, 2.5);
```

For the structure version you can use the `{...}` syntax. However, for the class version you need to call the constructor using function notation `(...)`.

Since C++11, you can more uniformly use the `{...}` syntax to initialize types, as follows:

```
CircleStruct myCircle3 = {10, 10, 2.5};
CircleClass myCircle4 = {10, 10, 2.5};
```

The definition of `myCircle4` will automatically call the constructor of `CircleClass`. The use of the equal sign is even optional, so the following is identical:

```
CircleStruct myCircle5{10, 10, 2.5};
CircleClass myCircle6{10, 10, 2.5};
```

Uniform initialization is not limited to structures and classes. You can use it to initialize anything in C++. For example, the following code initializes all four variables with the value 3:

```
int a = 3;
int b(3);
```

```
int c = {3}; // Uniform initialization
int d{3}; // Uniform initialization
```

Uniform initialization can be used to perform zero-initialization of variables; you just specify an empty set of curly braces. For example:

```
int e{}; // Uniform initialization, e will be 0
```

Using uniform initialization also prevents *narrowing*. C++ implicitly performs narrowing; for example:

```
void func(int i) { /* ... */ }
int main()
{
    int x = 3.14;
    func(3.14);
    return 0;
}
```

C++ will automatically truncate 3.14 in both cases to 3 before assigning it to x or calling func(). Note that some compilers **might** issue a warning about this narrowing, and some compilers won't give a warning. However, you can use uniform initialization as follows:

```
void func(int i) { /* ... */ }
int main()
{
    int x = {3.14}; // Error or warning because narrowing
    func({3.14}); // Error or warning because narrowing
    return 0;
}
```

Now both the assignment to x and the call to func() **must** generate a compiler error or warning if your compiler fully conforms to the C++11 standard.

Uniform initialization can also be used on STL containers, which are discussed in depth in Chapter 16. For example, initializing a vector of strings used to require code as follows:

```
std::vector<std::string> myVec;
myVec.push_back("String 1");
myVec.push_back("String 2");
myVec.push_back("String 3");
```

With uniform initialization, this can be rewritten:

```
std::vector<std::string> myVec = {"String 1", "String 2", "String 3"};
```

Uniform initialization can also be used to initialize dynamically allocated arrays. For example:

```
int* pArray = new int[4]{0, 1, 2, 3};
```

And last but not least, it can also be used in the constructor initializer to initialize arrays that are members of a class.

```

class MyClass
{
public:
    MyClass() : mArray{0, 1, 2, 3} {}
private:
    int mArray[4];
};

```

Initializer Lists

Initializer lists are defined in the `<initializer_list>` header file. They make it easy to write functions that can accept a variable number of arguments. The difference with variable-length argument lists described later in this chapter is that all the elements in an initializer list should have the same predefined type. The following example shows how to use an initializer list:

```

#include <initializer_list>
using namespace std;
int makeSum(initializer_list<int> lst)
{
    int total = 0;
    for (const auto& value : lst) {
        total += value;
    }
    return total;
}

```

The function `makeSum()` accepts an initializer list of integers as argument. The body of the function uses a range-based `for` loop to accumulate the total sum. This function can be used as follows:

```

int a = makeSum({1,2,3});
int b = makeSum({10,20,30,40,50,60});

```

Initializer lists are type-safe and define which type is allowed to be in the list. For the above `makeSum()` function, you could call it with a `double` value as follows:

```

int c = makeSum({1,2,3.0});

```

The last element is a `double`, which will result in a compiler error or warning.

Explicit Conversion Operators

Chapter 8 discusses the implicit conversion that can happen with single argument constructors and how to prevent the compiler from using those implicit conversions with the `explicit` keyword. The C++ compiler will also perform implicit conversion with custom written conversion operators.

Since C++11 it is possible to apply the `explicit` keyword, not only to constructors, but also to conversion operators.

To explain explicit conversion operators, you need to understand implicit conversion first. Take the following example. It defines a class `IntWrapper` that just wraps an integer and implements an `int()` conversion operator, which the compiler can use to perform implicit conversion from an `IntWrapper` to type `int`.

```
class IntWrapper
{
public:
    IntWrapper(int i) : mInt(i) {}
    operator int() const { return mInt; }
private:
    int mInt;
};
```

The following code demonstrates this implicit conversion; `iC1` will contain the value 123:

```
IntWrapper c(123);
int iC1 = c;
```

If you want, you can still explicitly tell the compiler to call the `int()` conversion operator as follows. `iC2` will also contain the value 123.

```
int iC2 = static_cast<int>(c);
```

Since C++11 you can use the `explicit` keyword to prevent the compiler from performing the implicit conversion. Below is the new class definition:

```
class IntWrapper
{
public:
    IntWrapper(int i) : mInt(i) {}
    explicit operator int() const { return mInt; }
private:
    int mInt;
};
```

Trying to compile the following lines of code with this new class definition will result in a compiler error because the `int()` conversion operator is marked as `explicit`, so the compiler cannot use it anymore to perform implicit conversions.

```
IntWrapper c(123);
int iC1 = c; // Error, because of explicit int() operator
```

Once you have an explicit conversion operator, you have to explicitly call it if you want to use it. For example:

```
int iC2 = static_cast<int>(c);
```

Attributes

Attributes are a mechanism to add optional and/or vendor-specific information into source code. Before C++11, the vendor decided how to specify that information. Examples are `__attribute__`, `__declspec`, and so on. Since C++11, there is support for attributes by using the double square brackets syntax `[[attribute]]`.

The C++11 standard defines only two standard attributes: `[[noreturn]]` and `[[carries_dependency]]`. C++14 adds the `[[deprecated]]` attribute.

`[[noreturn]]` means that a function never returns control to the call site. Typically, the function either causes some kind of termination (process termination or thread termination) or throws an exception. For example:

```
[[noreturn]] void func()
{
    throw 1;
}
```

The second attribute, `[[carries_dependency]]`, is a rather exotic attribute and is not discussed further.

`[[deprecated]]` can be used to mark something as deprecated, which means you can still use it, but its use is discouraged. This attribute accepts an optional argument that can be used to explain the reason of the deprecation, for example `[[deprecated("Unsafe method, please use xyz")]]`.

Most attributes will be vendor-specific extensions. Vendors are advised not to use attributes to change the meaning of the program, but to use them to help the compiler to optimize code or detect errors in code. Since attributes of different vendors could clash, vendors are recommended to qualify them. For example:

```
[[clang::noduplicate]]
```

User-Defined Literals

C++ has a number of standard literals that you can use in your code. For example:

- `'a'`: character
- `"character array"`: zero-terminated array of characters, C-style string
- `3.14f`: float floating point value
- `0xabc`: hexadecimal value

C++11 allows you to define your own literals. These user-defined literals should start with an underscore and are implemented by writing *literal operators*. A literal operator can work in *raw* or *cooked* mode. In raw mode, your literal operator receives a sequence of characters; while in cooked mode your literal operator receives a specific interpreted type. For example, take the C++ literal `123`. Your raw literal operator receives this as a sequence of characters `'1'`, `'2'`, `'3'`. Your cooked literal operator receives this as the integer `123`. Another example, take the C++ literal `0x23`. The raw operator receives the characters `'0'`, `'x'`, `'2'`, `'3'`, while the cooked operator receives the integer `35`. One last example, take the C++ literal `3.14`. Your raw operator receives this as `'3'`, `'.'`, `'1'`, `'4'`, while your cooked operator receives the floating point value `3.14`.

A cooked-mode literal operator should have:

- one parameter of type `unsigned long long`, `long double`, `char`, `wchar_t`, `char16_t`, or `char32_t` to process numeric values,
- or two parameters where the first is a character array and the second is the length of the character array, to process strings. For example: `const char* str, size_t len`.

As an example, the following implements a cooked literal operator for the user-defined literal `_i` to define a complex number literal:

```
std::complex<double> operator"" _i(long double d)
{
    return std::complex<double>(0, d);
}
```

This `_i` literal can be used as follows:

```
std::complex<double> c1 = 9.634_i;
auto c2 = 1.23_i;           // c2 will have as type std::complex<double>
```

A second example implements a cooked operator for a user-defined literal `_s` to define `std::string` literals:

```
std::string operator"" _s(const char* str, size_t len)
{
    return std::string(str, len);
}
```

This literal can be used as follows:

```
std::string str1 = "Hello World"_s;
auto str2 = "Hello World"_s;    // str2 will have as type std::string
```

Without the `_s` literal, the `auto` type deduction would be `const char*`:

```
auto str3 = "Hello World";      // str3 will have as type const char*
```

A raw-mode literal operator requires one parameter of type `const char*`; a zero-terminated C-style string. The following example defines the literal `_i` but using a raw literal operator:

```
std::complex<double> operator"" _i(const char* p)
{
    // Implementation omitted; it requires parsing the C-style
    // string and converting it to a complex number.
}
```

Using this raw mode operator is exactly the same as using the cooked version.

Standard User-Defined Literals

C++14 defines the following standard user-defined literals:

- “s” for creating `std::strings`; for example:


```
auto myString = "Hello World"s;
```
- “h”, “min”, “s”, “ms”, “us”, “ns”, for creating `std::chrono::duration` time intervals, discussed in Chapter 19; for example:


```
auto myDuration = 42min;
```

- “i”, “il”, “if” for creating complex numbers `complex<double>`, `complex<long double>`, and `complex<float>` respectively; for example:
`auto myComplexNumber = 1.3i;`

HEADER FILES

Header files are a mechanism for providing an abstract interface to a subsystem or piece of code. One of the trickier parts of using headers is avoiding circular references and multiple includes of the same header file. For example, perhaps you are responsible for writing the `Logger` class that performs all error message logging tasks. You may end up using another class, `Preferences`, that keeps track of user settings. The `Preferences` class may in turn use the `Logger` class indirectly, through yet another header.

As the following code shows, the `#ifndef` mechanism can be used to avoid circular and multiple includes. At the beginning of each header file, the `#ifndef` directive checks to see if a certain key has *not* been defined. If the key has been defined, the compiler will skip to the matching `#endif`, which is usually placed at the end of the file. If the key has *not* been defined, the file will proceed to define the key so that a subsequent include of the same file will be skipped. This mechanism is also known as *include guards*.

```
#ifndef LOGGER_H
#define LOGGER_H
#include "Preferences.h"
class Logger
{
public:
    static void setPreferences(const Preferences& prefs);
    static void logError(const char* error);
};
#endif // LOGGER_H
```

If your compiler supports the `#pragma once` directive (like Microsoft Visual C++ or GCC), this can be rewritten as follows:

```
#pragma once
#include "Preferences.h"
class Logger
{
public:
    static void setPreferences(const Preferences& prefs);
    static void logError(const char* error);
};
```

These include guards or `#pragma once` directives also make sure that you don't get duplicate definitions by including a header file multiple times. For example, suppose `A.h` includes `Logger.h` and `B.h` also includes `Logger.h`. If you have a source file called `App.cpp`, which includes both `A.h` and `B.h`, the compiler will not complain about a duplicate definition of the `Logger` class because the `Logger.h` header will be included only once, even though `A.h` and `B.h` both include it.

Another tool for avoiding problems with headers is forward declarations. If you need to refer to a class but you cannot include its header file (for example, because it relies heavily on the class you are writing), you can tell the compiler that such a class exists without providing a formal definition through the `#include` mechanism. Of course, you cannot actually use the class in the code because the compiler knows nothing about it, except that the named class will exist after everything is linked together. However, you can still make use of pointers or references to the class in your code. In the following code, the `Logger` class refers to the `Preferences` class without including its header file:

```
#ifndef LOGGER_H
#define LOGGER_H
class Preferences; // forward declaration
class Logger
{
public:
    static void setPreferences(const Preferences& prefs);
    static void logError(const char* error);
};
#endif // LOGGER_H
```

It's recommended to use forward declarations as much as possible in your header files instead of including other headers. This can reduce your compilation and recompilation times, because it breaks dependencies of your header file on other headers. Of course, your implementation file needs to include the correct headers for types that you've forward-declared, otherwise it won't compile.

C UTILITIES

There are a few obscure C features that are also available in C++ and which can occasionally be useful. This section examines two of these features: variable-length argument lists and preprocessor macros.

Variable-Length Argument Lists

This section explains the old C-style variable-length argument lists. You need to know how these work because you might find them in older code. However, in new code you should use variadic templates for type-safe variable-length argument lists, described in Chapter 21.

Consider the C function `printf()` from `<cstdio>`. You can call it with any number of arguments:

```
printf("int %d\n", 5);
printf("String %s and int %d\n", "hello", 5);
printf("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);
```

C/C++ provides the syntax and some utility macros for writing your own functions with a variable number of arguments. These functions usually look a lot like `printf()`. Although you shouldn't need this feature very often, occasionally you run into situations in which it's quite useful. For example, suppose you want to write a quick-and-dirty debug function that prints strings to `stderr` if a debug flag is set, but does nothing if the debug flag is not set. Just as `printf()`, this function

should be able to print strings with arbitrary numbers of arguments and arbitrary types of arguments. A simple implementation looks like this:

```
#include <cstdio>
#include <cstdarg>
bool debug = false;
void debugOut(const char* str, ...)
{
    va_list ap;
    if (debug) {
        va_start(ap, str);
        vfprintf(stderr, str, ap);
        va_end(ap);
    }
}
```

First, note that the prototype for `debugOut()` contains one typed and named parameter `str`, followed by `...` (ellipses). They stand for any number and type of arguments. In order to access these arguments, you must use macros defined in `<cstdarg>`. You declare a variable of type `va_list`, and initialize it with a call to `va_start`. The second parameter to `va_start()` must be the rightmost *named* variable in the parameter list. All functions with variable-length argument lists require at least one named parameter. The `debugOut()` function simply passes this list to `vfprintf()` (a standard function in `<cstdio>`). After the call to `vfprintf()` returns, `debugOut()` calls `va_end()` to terminate the access of the variable argument list. You must always call `va_end()` after calling `va_start()` to ensure that the function ends with the stack in a consistent state.

You can use the function in the following way:

```
debug = true;
debugOut("int %d\n", 5);
debugOut("String %s and int %d\n", "hello", 5);
debugOut("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);
```

Accessing the Arguments

If you want to access the actual arguments yourself, you can use `va_arg()` to do so. Unfortunately, there is no way to know what the end of the argument list is unless you provide an explicit way of doing so. For example, you can make the first parameter a count of the number of parameters. Or, in the case where you have a set of pointers, you may require the last pointer to be `nullptr`. There are many ways, but they are all burdensome to the programmer.

The following example demonstrates the technique where the caller specifies in the first named parameter how many arguments are provided. The function accepts any number of `ints` and prints them out:

```
void printInts(int num, ...)
{
    int temp;
    va_list ap;
    va_start(ap, num);
    for (int i = 0; i < num; ++i) {
        temp = va_arg(ap, int);
```

```

        cout << temp << " ";
    }
    va_end(ap);
    cout << endl;
}

```

You can call `printInts()` as follows. Note that the first parameter specifies how many integers will follow:

```
printInts(5, 5, 4, 3, 2, 1);
```

Why You Shouldn't Use C-Style Variable-Length Argument Lists

Accessing C-style variable-length argument lists is not very safe. There are several risks, as you can see from the `printInts()` function:

- You don't know the number of parameters. In the case of `printInts()`, you must trust the caller to pass the right number of arguments as the first argument. In the case of `debugOut()`, you must trust the caller to pass the same number of arguments after the character array as there are formatting codes in the character array.
- You don't know the types of the arguments. `va_arg()` takes a type, which it uses to interpret the value in its current spot. However, you can tell `va_arg()` to interpret the value as any type. There is no way for it to verify the correct type.

WARNING *Avoid using C-style variable-length argument lists. It is preferable to pass in an `std::array` or `vector` of values, to use initializer lists described earlier in this chapter, or to use variadic templates for type-safe variable-length argument lists described in Chapter 21.*

Preprocessor Macros

You can use the C++ preprocessor to write *macros*, which are like little functions. Here is an example:

```
#define SQUARE(x) ((x) * (x)) // No semicolon after the macro definition!
int main()
{
    cout << SQUARE(5) << endl;
    return 0;
}
```

Macros are a remnant from C that are quite similar to `inline` functions, except that they are not type-checked, and the preprocessor dumbly replaces any calls to them with their expansions. The preprocessor does not apply true function-call semantics. This behavior can cause unexpected results. For example, consider what would happen if you called the `SQUARE` macro with `2 + 3` instead of `5`, like this:

```
cout << SQUARE(2 + 3) << endl;
```

You expect `SQUARE` to calculate 25, which it does. However, what if you left off some parentheses on the macro definition, so that it looks like this?

```
#define SQUARE(x) (x * x)
```

Now, the call to `SQUARE(2 + 3)` generates 11, not 25! Remember that the macro is dumbly expanded without regard to function-call semantics. This means that any `x` in the macro body is replaced by `2 + 3`, leading to this expansion:

```
cout << (2 + 3 * 2 + 3) << endl;
```

Following proper order of operations, this line performs the multiplication first, followed by the additions, generating 11 instead of 25!

Macros can also have a performance impact. Suppose you call the `SQUARE` macro as follows:

```
cout << SQUARE(veryExpensiveFunctionCallToComputeNumber()) << endl;
```

The preprocessor replaces this with:

```
cout << ((veryExpensiveFunctionCallToComputeNumber()) *  
         (veryExpensiveFunctionCallToComputeNumber())) << endl;
```

Now you are calling the expensive function twice. Another reason to avoid macros.

Macros also cause problems for debugging because the code you write is not the code that the compiler sees, or that shows up in your debugger (because of the search-and-replace behavior of the preprocessor). For these reasons, you should avoid macros entirely in favor of inline functions. The details are shown here only because quite a bit of C++ code out there employs macros. You need to understand them in order to read and maintain that code.

NOTE *Some compilers can output the preprocessed source to a file. You can use that file to see how the preprocessor is preprocessing your file. For example, with Microsoft VC++ you need to use the /P switch. With GCC you can use the -E switch.*

SUMMARY

This chapter explained some of the aspects of C++ that generate confusion. By reading this chapter, you learned a plethora of syntax details about C++. Some of the information, such as the details of references, `const`, scope resolution, the specifics of the C++-style casts, and the techniques for header files, you should use often in your programs. Other information, such as the uses of `static` and `extern`, how to write C-style variable-length argument lists, and how to write preprocessor macros, is important to understand, but not information that you should put into use in your programs on a day-to-day basis.

This chapter also discussed a number of C++11 and C++14 features that don't really fit anywhere else in the book.

The next chapter starts a discussion on templates allowing you to write generic code.

11

Writing Generic Code with Templates

WHAT'S IN THIS CHAPTER?

- How to write class templates
- How the compiler processes templates
- How to organize template source code
- How to use non-type template parameters
- How to write templates of individual class methods
- How to write customizations of your class templates for specific types
- How to combine templates and inheritance
- How to write function templates
- How to make function templates friends of class templates
- How to write alias templates
- Variable templates defined

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

C++ provides language support not only for object-oriented programming, but also for *generic programming*. As discussed in Chapter 6, the goal of generic programming is to write reusable code. The fundamental tools for generic programming in C++ are *templates*. Although not strictly an object-oriented feature, templates can be combined with object-oriented programming for powerful results. Many programmers consider templates to be the most difficult part of C++ and, for that reason, tend to avoid them.

This chapter provides the code details for fulfilling the design principle of generality discussed in Chapter 6, while Chapter 21 delves into some of the more advanced template features, including the following:

- The three kinds of template parameters and their subtleties
- Partial specialization
- Function template deduction
- How to exploit template recursion
- Variadic templates
- Metaprogramming

OVERVIEW OF TEMPLATES

The main programming unit in the procedural paradigm is the *procedure* or *function*. Functions are useful primarily because they allow you to write algorithms that are independent of specific values and can thus be reused for many different values. For example, the `sqrt()` function in C++ calculates the square root of a value supplied by the caller. A square root function that calculates only the square root of one number, such as four, would not be particularly useful! The `sqrt()` function is written in terms of a *parameter*, which is a stand-in for whatever value the caller passes. Computer scientists say that functions *parameterize* values.

The object-oriented programming paradigm adds the concept of *objects*, which group related data and behaviors, but does not change the way functions and methods parameterize values.

Templates take the concept of parameterization a step further to allow you to parameterize on *types* as well as *values*. Types in C++ include primitives such as `int` and `double`, as well as user-defined classes such as `SpreadsheetCell` and `cherryTree`. With templates you can write code that is independent not only of the values it will be given, but also of the types of those values. For example, instead of writing separate stack classes to store `ints`, `Cars`, and `SpreadsheetCells`, you can write one stack class definition that can be used for any of those types.

Although templates are an amazing language feature, templates in C++ are syntactically confusing, and many programmers overlook or avoid them.

This chapter teaches you about template support in C++ with an emphasis on the aspects that arise in the Standard Template Library (STL). Along the way, you learn about some nifty features that you can employ in your programs aside from using the standard library.

CLASS TEMPLATES

Class templates define a class where the types of some of the variables, return types of methods, and/or parameters to the methods are specified as parameters. Class templates are useful primarily for containers, or data structures, that store objects. This section uses a running example of a `Grid` container. In order to keep the examples reasonable in length and simple enough to illustrate specific points, different sections of the chapter add features to the `Grid` container that are not used in subsequent sections.

Writing a Class Template

Suppose that you want a generic game board class that you can use as a chessboard, checkers board, Tic-Tac-Toe board, or any other two-dimensional game board. In order to make it general-purpose, you should be able to store chess pieces, checkers pieces, Tic-Tac-Toe pieces, or any type of game piece.

Coding without Templates

Without templates, the best approach to build a generic game board is to employ polymorphism to store generic `GamePiece` objects. Then, you could let the pieces for each game inherit from the `GamePiece` class. For example, in a chess game, `ChessPiece` would be a derived class of `GamePiece`. Through polymorphism, the `GameBoard`, written to store `GamePieces`, can also store `ChessPieces`. Because it should be possible to copy a `GameBoard`, the `GameBoard` needs to be able to copy `GamePieces`. This implementation employs polymorphism, so one solution is to add a pure virtual `clone()` method to the `GamePiece` base class. The basic `GamePiece` interface is:

```
class GamePiece
{
public:
    virtual std::unique_ptr<GamePiece> clone() const = 0;
};
```

`GamePiece` is an abstract base class. Concrete classes, such as `ChessPiece`, derive from it and implement the `clone()` method:

```
class ChessPiece : public GamePiece
{
public:
    virtual std::unique_ptr<GamePiece> clone() const override;
};

std::unique_ptr<GamePiece> ChessPiece::clone() const
{
    // Call the copy constructor to copy this instance
    return std::make_unique<ChessPiece>(*this);
}
```

The implementation of `GameBoard` uses a vector of vectors of `unique_ptr`s to store the `GamePieces`.

```
class GameBoard
{
public:
```

```

        explicit GameBoard(size_t inWidth = kDefaultWidth,
                            size_t inHeight = kDefaultHeight);
        GameBoard(const GameBoard& src); // copy constructor
        virtual ~GameBoard();
        GameBoard& operator=(const GameBoard& rhs); // assignment operator

        // Sets a piece at a location. The GameBoard becomes owner of the piece.
        // inPiece can be nullptr to remove any piece from the given location.
        void setPieceAt(size_t x, size_t y, std::unique_ptr<GamePiece> inPiece);
        std::unique_ptr<GamePiece>& getPieceAt(size_t x, size_t y);
        const std::unique_ptr<GamePiece>& getPieceAt(size_t x, size_t y) const;

        size_t getHeight() const { return mHeight; }
        size_t getWidth() const { return mWidth; }
        static const size_t kDefaultWidth = 10;
        static const size_t kDefaultHeight = 10;

    private:
        void copyFrom(const GameBoard& src);
        void initializeCellsContainer();
        std::vector<std::vector<std::unique_ptr<GamePiece>>> mCells;
        size_t mWidth, mHeight;
    };

```

The implementation of `setPieceAt()` moves the given piece into `mCells`. The `GameBoard` becomes the owner of the piece.

`getPieceAt()` returns a reference to the piece at a specified spot instead of a copy of the piece. The `GameBoard` serves as an abstraction of a two-dimensional array, so it should provide array access semantics by giving the actual object at an index, not a copy of the object. Client code should not store this reference for future use because it might become invalid. Instead, client code should call `getPieceAt()` right before using the returned reference.

NOTE *This implementation of the class provides two versions of `getPieceAt()`, one of which returns a reference and one of which returns a const reference.*

Here are the method definitions. Production code would, of course, perform bounds checking in `setPieceAt()` and `getPieceAt()`. That code is omitted because it is not the point of this chapter:

```

GameBoard::GameBoard(size_t inWidth, size_t inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    initializeCellsContainer();
}

GameBoard::GameBoard(const GameBoard& src)
{
    copyFrom(src);
}

```

```
}

GameBoard::~GameBoard()
{
    // Nothing to do, the vector will clean up itself.
}

void GameBoard::initializeCellsContainer()
{
    mCells.resize(mWidth);
    for (auto& column : mCells) {
        column.resize(mHeight);
    }
}

void GameBoard::copyFrom(const GameBoard& src)
{
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    initializeCellsContainer();
    for (size_t i = 0; i < mWidth; i++) {
        for (size_t j = 0; j < mHeight; j++) {
            if (src.mCells[i][j])
                mCells[i][j] = src.mCells[i][j]->clone();
            else
                mCells[i][j].reset();
        }
    }
}

GameBoard& GameBoard::operator=(const GameBoard& rhs)
{
    // check for self-assignment
    if (this == &rhs) {
        return *this;
    }
    // copy the source GameBoard
    copyFrom(rhs);
    return *this;
}

void GameBoard::setPieceAt(size_t x, size_t y, unique_ptr<GamePiece> inPiece)
{
    mCells[x][y] = move(inPiece);
}

unique_ptr<GamePiece>& GameBoard::getPieceAt(size_t x, size_t y)
{
    return mCells[x][y];
}

const unique_ptr<GamePiece>& GameBoard::getPieceAt(size_t x, size_t y) const
{
    return mCells[x][y];
}
```

This GameBoard class works pretty well:

```
GameBoard chessBoard(8, 8);
auto pawn = std::make_unique<ChessPiece>();
chessBoard.setPieceAt(0, 0, std::move(pawn));
chessBoard.setPieceAt(0, 1, std::make_unique<ChessPiece>());
chessBoard.setPieceAt(0, 1, nullptr);
```

A Template Grid Class

The GameBoard class in the previous section is nice, but insufficient. For example, it's quite similar to the Spreadsheet class from Chapter 8, but the only way you could use it as a spreadsheet would be to make the SpreadsheetCell class derive from GamePiece. That doesn't make sense because it doesn't fulfill the is-a principle of inheritance: a SpreadsheetCell is not a GamePiece. It would be nice if you could write a generic Grid class that you could use for purposes as diverse as a Spreadsheet or a ChessBoard. In C++, you can do this by writing a *class template*, which allows you to write a class without specifying one or more types. Clients then *instantiate* the template by specifying the types they want to use. This is called *generic programming*. The biggest advantage of generic programming is type safety. The types used in the class and its methods are concrete types, and not abstract base class types as is the case with the polymorphic solution from the previous section.

The Grid Class Definition

In order to understand class templates, it is helpful to examine the syntax. The following example shows how you can modify your GameBoard class to make a templatized Grid class. The syntax is explained in detail following the code. Note that the class name has changed from GameBoard to Grid, and setPieceAt() and getPieceAt() have changed to setElementAt() and getElementAt() to reflect the class's more generic nature. I also want this Grid to be useable with primitive types such as int and double. That's why I opted to implement this solution using value semantics without polymorphism, compared to the polymorphic pointer semantics used in the GameBoard implementation.

```
template <typename T>
class Grid
{
public:
    explicit Grid(size_t inWidth = kDefaultWidth,
                  size_t inHeight = kDefaultHeight);
    virtual ~Grid();

    // Sets an element at a given location. The element is copied.
    void setElementAt(size_t x, size_t y, const T& inElem);
    T& getElementAt(size_t x, size_t y);
    const T& getElementAt(size_t x, size_t y) const;

    size_t getHeight() const { return mHeight; }
    size_t getWidth() const { return mWidth; }
    static const size_t kDefaultWidth = 10;
    static const size_t kDefaultHeight = 10;

private:
```

```

    void initializeCellsContainer();
    std::vector<std::vector<T>> mCells;
    size_t mWidth, mHeight;
};

}

```

Now that you've seen the full class definition, take another look at it, one line at a time:

```
template <typename T>
```

This first line says that the following class definition is a template on one type. Both `template` and `typename` are keywords in C++. As discussed earlier, templates “parameterize” types the same way that functions “parameterize” values. Just as you use parameter names in functions to represent the arguments that the caller will pass, you use template parameter names (such as `T`) in templates to represent the types that the caller will specify. There's nothing special about the name `T` — you can use whatever name you want. Traditionally, when a single type is used, it is called `T`, but that's just a historical convention, like calling the integer that indexes an array `i` or `j`. The template specifier holds for the entire statement, which in this case is the class definition.

NOTE *For historical reasons, you can use the keyword `class` instead of `typename` to specify template type parameters. Thus, many books and existing programs use syntax like this: `template <class T>`. However, the use of the word “class” in this context is confusing because it implies that the type must be a class, which is not true. The type can be a class, a struct, a union, a primitive type of the language like `int` or `double`, and so on.*

In the earlier `GameBoard` class, the `mCells` data member is a vector of vectors of *pointers*, which requires special code for copying, thus the need for a copy constructor and assignment operator. In the `Grid` class, `mCells` is a vector of vectors of *values*, so the default compiler-generated copy constructor and assignment operator are fine. However, if you add other data members to the `Grid` class that require you to add a copy constructor and assignment operator, then the syntax for them is as follows:

```

Grid(const Grid<T>& src);
Grid<T>& operator=(const Grid<T>& rhs);

```

As you can see, the type of the `src` parameter is no longer a `const GameBoard&`, but a `const Grid<T>&`. When you write a class template, what you used to think of as the class name (`Grid`) is actually the *template name*. When you want to talk about actual `Grid` classes or types, you discuss them as *instantiations* of the `Grid` class template for a certain type, such as `int`, `SpreadsheetCell`, or `ChessPiece`. At the point where you define the template, you haven't specified the type it will be instantiated for, so you must use a stand-in template parameter, `T`, for whatever type might be used later. Thus, when you need to refer to a type for a `Grid` object as a method's parameter or return value, you must use `Grid<T>`.

Within a class definition, the compiler will interpret `Grid` as `Grid<T>` where needed. However, it's best to get in the habit of specifying `Grid<T>` explicitly because that's the syntax you use outside the class to refer to types generated from the template. Only for constructors and the destructor, you should use `Grid` and not `Grid<T>`.

Because `mCells` is not storing pointers anymore, the `setElementAt()` method now takes a `const T&` as parameter instead of a `unique_ptr<GamePiece>`. The `getElementAt()` methods now return `T&` instead of `unique_ptr<GamePiece>&`:

```
void setElementAt(size_t x, size_t y, const T& inElem);
T& getElementAt(size_t x, size_t y);
const T& getElementAt(size_t x, size_t y) const;
```

The Grid Class Method Definitions

The template `<typename T>` specifier must precede each method definition for the `Grid` template. The constructor looks like this:

```
template <typename T>
Grid<T>::Grid(size_t inWidth, size_t inHeight) : mWidth(inWidth), mHeight(inHeight)
{
    initializeCellsContainer();
}
```

NOTE *Templates require you to put the implementation of the methods in the header file itself, because the compiler needs to know the complete definition, including the definition of methods, before it can create an instance of the template.*

Note that the class name before the `::` is `Grid<T>`, not `Grid`. You must specify `Grid<T>` as the class name in all your methods and static data member definitions. The body of the constructor is identical to the `GameBoard` constructor.

The rest of the method definitions are also similar to their equivalents in the `GameBoard` class with the exception of the appropriate `template` and `Grid<T>` syntax changes:

```
template <typename T>
Grid<T>::~Grid()
{
    // Nothing to do, the vector will clean up itself.
}

template <typename T>
void Grid<T>::initializeCellsContainer()
{
    mCells.resize(mWidth);
    for (std::vector<T>& column : mCells) {
        column.resize(mHeight);
    }
}

template <typename T>
void Grid<T>::setElementAt(size_t x, size_t y, const T& inElem)
{
    mCells[x][y] = inElem;
}

template <typename T>
```

```

T& Grid<T>::getElementAt(size_t x, size_t y)
{
    return mCells[x][y];
}

template <typename T>
const T& Grid<T>::getElementAt(size_t x, size_t y) const
{
    return mCells[x][y];
}

```

Using the Grid Template

When you want to create grid objects, you cannot use `Grid` alone as a type; you must specify the type that will be stored in that `Grid`. Creating an object of a class template for a specific type is called *instantiating the template*. Here is an example:

```

Grid<int> myIntGrid; // declares a grid that stores ints,
                     // using default parameters for the constructor
Grid<double> myDoubleGrid(11, 11); // declares an 11x11 Grid of doubles
myIntGrid.setElementAt(0, 0, 10);
int x = myIntGrid.getElementAt(0, 0);
Grid<int> grid2(myIntGrid); // Copy constructor
Grid<int> anotherIntGrid;
anotherIntGrid = grid2;      // Assignment operator

```

Note that the type of `myIntGrid`, `grid2`, and `anotherIntGrid` is `Grid<int>`. You cannot store `SpreadsheetCells` or `ChessPieces` in these grids; the compiler will generate an error if you try to do so.

The type specification is important; neither of the following two lines compiles:

```

Grid test; // WILL NOT COMPILE
Grid<> test; // WILL NOT COMPILE

```

The first causes your compiler to complain with something like, “use of class template requires template argument list.” The second causes it to say something like, “too few template arguments.”

If you want to declare a function or method that takes a `Grid` object, you must specify the type stored in that grid as part of the `Grid` type:

```

void processIntGrid(Grid<int>& inGrid)
{
    // Body omitted for brevity
}

```

NOTE *Instead of writing the full Grid type every time, for example `Grid<int>`, you can use a type alias to give it an easier name:*

```
using IntGrid = Grid<int>;
```

Now you can write code as follows:

```
void processIntGrid(IntGrid& inGrid) { }
```

The `Grid` template can store more than just `ints`. For example, you can instantiate a `Grid` that stores `SpreadsheetCells`:

```
Grid<SpreadsheetCell> mySpreadsheet;
SpreadsheetCell myCell("Test");
mySpreadsheet.setElementAt(3, 4, myCell);
```

You can store pointer types as well:

```
Grid<const char*> myStringGrid;
myStringGrid.setElementAt(2, 2, "hello");
```

The type specified can even be another template type:

```
Grid<vector<int>> gridOfVectors;
vector<int> myVector{ 1, 2, 3, 4 };
gridOfVectors.setElementAt(5, 6, myVector);
```

You can also dynamically allocate `Grid` template instantiations on the heap:

```
auto myGridp = make_unique<Grid<int>>(2, 2); // creates a 2x2 Grid on the heap
myGridp->setElementAt(0, 0, 10);
int x = myGridp->getElementAt(0, 0);
```

Angle Brackets

Some of the examples in this book use templates with double angle brackets; for example:

```
std::vector<std::vector<T>> mCells;
```

This works perfectly fine since C++11. However, before C++11, the double angle brackets `>>` could mean only one thing: the `>>` operator. Depending on the types involved, this `>>` operator can be a right bit-shift operation, or a stream extraction operator. This was annoying with template code, because you were forced to put a space between double angle brackets. The previous declaration had to be written as follows:

```
std::vector<std::vector<T> > mCells;
```

This book uses the modern style without the spaces.

How the Compiler Processes Templates

In order to understand the intricacies of templates, you need to learn how the compiler processes template code. When the compiler encounters template method definitions, it performs syntax checking, but doesn't actually compile the templates. It can't compile template definitions because it doesn't know for which types they will be used. It's impossible for a compiler to generate code for something like `x = y` without knowing the types of `x` and `y`.

When the compiler encounters an instantiation of the template, such as `Grid<int> myIntGrid`, it writes code for an `int` version of the `Grid` template by replacing each `T` in the class template definition with `int`. When the compiler encounters a different instantiation of the template, such as `Grid<SpreadsheetCell> mySpreadsheet`, it writes another version of the `Grid` class for `SpreadsheetCells`. The compiler just writes the code that you would write if you didn't have template support in the language and had to write separate classes for each element type. There's no

magic here; templates just automate an annoying process. If you don't instantiate a class template for any types in your program, then the class method definitions are never compiled.

This instantiation process explains why you need to use the `Grid<T>` syntax in various places in your definition. When the compiler instantiates the template for a particular type, such as `int`, it replaces `T` with `int`, so that `Grid<int>` is the type.

Selective Instantiation

The compiler always generates code for all virtual methods of a generic class. However, for non-virtual methods, the compiler generates code only for those non-virtual methods that you actually call for a particular type. For example, given the preceding `Grid` class template, suppose that you write this code (and only this code) in `main()`:

```
Grid<int> myIntGrid;
myIntGrid.setElementAt(0, 0, 10);
```

The compiler generates only the 0-argument constructor, the destructor, and the `setElementAt()` method for an `int` version of the `Grid`. It does not generate other methods like the copy constructor, the assignment operator, or `getHeight()`.

Template Requirements on Types

When you write code that is independent of types, you must assume certain things about those types. For example, in the `Grid` template, you assume that the element type (represented by `T`) will have an assignment operator because of this line: `mCells[x][y] = inElem`. Similarly, you assume it will have a default constructor so you can create a vector of elements.

If you attempt to instantiate a template with a type that does not support all the operations used by the template in your particular program, the code will not compile, and the error messages will almost always be quite obscure. However, even if the type you want to use doesn't support the operations required by all the template code, you can exploit selective instantiation to use some methods but not others. For example, if you try to create a grid for an object that has no assignment operator, but you never call `setElementAt()` on that grid, your code will work fine. As soon as you try to call `setElementAt()`, however, you will receive a compilation error.

Distributing Template Code between Files

Normally you put class definitions in a header file and method definitions in a source file. Code that creates or uses objects of the class `#includes` the header file and obtains access to the method code via the linker. Templates don't work that way. Because they are "templates" for the compiler to generate the actual methods for the instantiated types, both class template definitions and method definitions must be available to the compiler in any source file that uses them. There are several mechanisms to obtain this inclusion.

Template Definitions in Header Files

You can place the method definitions directly in the same header file where you define the class itself. When you `#include` this file in a source file where you use the template, the compiler will have access to all the code it needs.

Alternatively, you can place the template method definitions in a separate header file that you `#include` in the header file with the class definitions. Make sure the `#include` for the method definitions follows the class definition; otherwise the code won't compile.

```
template <typename T>
class Grid
{
    // Class definition omitted for brevity
};
#include "GridDefinitions.h"
```

Any client that wants to use the `Grid` template needs only to include the `Grid.h` header file. This division helps keep the distinction between class definitions and method definitions.

Template Definitions in Source Files

Method implementations look strange in header files. If that syntax annoys you, there is a way that you can place the method definitions in a source file. However, you still need to make the definitions available to the code that uses the templates, which you can do by `#include`ing the method implementation *source* file in the class template definition header file. That sounds odd if you've never seen it before, but it's legal in C++. The header file looks like this:

```
template <typename T>
class Grid
{
    // Class definition omitted for brevity
};
#include "Grid.cpp"
```

When using this technique, make sure you don't add the `Grid.cpp` file to your project, because it is not supposed to be, and cannot be compiled separately; it should be `#included` only in a header file.

You can actually call your file with method implementations anything you want. Some programmers like to give source files that are included in an `.inl` extension; for example `Grid.inl`.

Limit Class Template Instantiations

If you want your class templates to be used only with certain known types, you can use the following technique.

Suppose you want the `Grid` class to be instantiated only for `int`, `double`, and `vector<int>`. The header file should look like this:

```
template <typename T>
class Grid
{
    // Class definition omitted for brevity
};
```

Note that there are no method definitions in this header file and that there is no `#include` at the end!

In this case, you need a real `.cpp` file added to your project, which contains the method definitions and looks as follows:

```

#include "Grid.h"
template <typename T>
Grid<T>::Grid(size_t inWidth, size_t inHeight) : mWidth(inWidth), mHeight(inHeight)
{
    initializeCellsContainer();
}
// Other method definitions omitted for brevity...

```

For this method to work, you need to explicitly instantiate the template for those types that you want to allow clients to use. At the end of the `.cpp` file you can do this as follows:

```

// Explicit instantiations for the types you want to allow.
template class Grid<int>;
template class Grid<double>;
template class Grid<std::vector<int>>;

```

With these explicit instantiations, you disallow client code from using the `Grid` class template with other types, such as `SpreadsheetCell`.

Template Parameters

In the `Grid` example, the `Grid` template has one *template parameter*: the type that is stored in the grid. When you write the class template, you specify the parameter list inside the angle brackets, like this:

```
template <typename T>
```

This parameter list is similar to the parameter list in a function or method. As in functions and methods, you can write a class with as many template parameters as you want. Additionally, these parameters don't have to be types, and they can have default values.

Non-Type Template Parameters

Non-type parameters are “normal” parameters such as `ints` and `pointers`: the kind of parameters with which you’re familiar from functions and methods. However, non-type template parameters can only be integral types (`char`, `int`, `long...`), enumeration types, pointers, and references.

In the `Grid` class template, you could use non-type template parameters to specify the height and width of the grid instead of specifying them in the constructor. The principle advantage to specifying non-type parameters in the template list instead of in the constructor is that the values are known before the code is compiled. Recall that the compiler generates code for templated methods by substituting in the template parameters before compiling. Thus, you can use a normal two-dimensional array in your implementation instead of a `vector` of `vectors` that is dynamically resized. Here is the new class definition:

```

template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid
{
public:
    Grid();
    virtual ~Grid();

    void setElementAt(size_t x, size_t y, const T& inElem);

```

```

T& getElementAt(size_t x, size_t y);
const T& getElementAt(size_t x, size_t y) const;

size_t getHeight() const { return HEIGHT; }
size_t getWidth() const { return WIDTH; }

private:
    T mCells[WIDTH][HEIGHT];
};


```

This class is even more compact than the previous version. Note that the template parameter list requires three parameters: the type of objects stored in the grid and the width and height of the grid. The width and height are used to create a two-dimensional array to store the objects. There is no need for a user-defined copy constructor or assignment operator. A default constructor is provided to zero-initialize the elements of `mCells` because the compiler-generated one does not zero-initialize it. Here are the class method definitions:

```

template <typename T, size_t WIDTH, size_t HEIGHT>
Grid<T, WIDTH, HEIGHT>::Grid() : mCells()          // Zero-initialize mCells
{
}

template <typename T, size_t WIDTH, size_t HEIGHT>
Grid<T, WIDTH, HEIGHT>::~Grid()
{
    // Nothing to do.
}

template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::setElementAt(size_t x, size_t y, const T& inElem)
{
    mCells[x][y] = inElem;
}

template <typename T, size_t WIDTH, size_t HEIGHT>
T& Grid<T, WIDTH, HEIGHT>::getElementAt(size_t x, size_t y)
{
    return mCells[x][y];
}

template <typename T, size_t WIDTH, size_t HEIGHT>
const T& Grid<T, WIDTH, HEIGHT>::getElementAt(size_t x, size_t y) const
{
    return mCells[x][y];
}

```

Note that wherever you previously specified `Grid<T>` you must now specify `Grid<T, WIDTH, HEIGHT>` to represent the three template parameters.

You can instantiate this template and use it like this:

```

Grid<int, 10, 10> myGrid;
Grid<int, 10, 10> anotherGrid;
myGrid.setElementAt(2, 3, 45);
anotherGrid = myGrid;
cout << anotherGrid.getElementAt(2, 3);

```

This code seems great. Despite the slightly messy syntax for declaring a `Grid`, the actual `Grid` code is a lot simpler. Unfortunately, there are more restrictions than you might think at first. First, you can't use a non-constant integer to specify the height or width. The following code doesn't compile:

```
size_t height = 10;
Grid<int, 10, height> testGrid; // DOES NOT COMPILE
```

However, if you define `height` as a constant, it compiles:

```
const size_t height = 10;
Grid<int, 10, height> testGrid; // compiles and works
```

`constexpr` functions with the correct return type also work. For example, if you have a `constexpr` function that returns a `size_t`, you can use it to initialize the `height` template parameter:

```
constexpr size_t getHeight() { return 10; }
...
Grid<double, 2, getHeight()> myDoubleGrid;
```

A second restriction might be more significant. Now that the `width` and `height` are template parameters, they are part of the type of each grid. That means that `Grid<int, 10, 10>` and `Grid<int, 10, 11>` are two different types. You can't assign an object of one type to an object of the other, and variables of one type can't be passed to functions or methods that expect variables of another type.

NOTE *Non-type template parameters become part of the type specification of instantiated objects.*

Default Values for Type Parameters

If you continue the approach of making `height` and `width` template parameters, you might want to provide defaults for the `height` and `width` non-type template parameters just as you did previously in the constructor of the `Grid<T>` class. C++ allows you to provide defaults for template parameters with a similar syntax. While you are at it, you could also provide a default for the `T` type parameter. Here is the class definition:

```
template <typename T = int, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid
{
    // Remainder is identical to the previous version
};
```

You need not specify the default values for `T`, `WIDTH`, and `HEIGHT` in the template specification for the method definitions. For example, here is the implementation of `setElementAt()`:

```
template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::setElementAt(size_t x, size_t y, const T& inElem)
{
    mCells[x][y] = inElem;
}
```

Now, you can instantiate a `Grid` without any template parameters, with only the element type, the element type and the width, or the element type, width, and height:

```
Grid<> myIntGrid;
Grid<int> myGrid;
Grid<int, 5> anotherGrid;
Grid<int, 5, 5> aFourthGrid;
```

The rules for default parameters in template parameter lists are the same as for functions or methods: you can provide defaults for parameters in order starting from the right.

Method Templates

C++ allows you to template individual methods of a class. These methods can be inside a class template or in a non-templatized class. When you write a templatized class method, you are actually writing many different versions of that method for many different types. Method templates come in useful for assignment operators and copy constructors in class templates.

WARNING *Virtual methods and destructors cannot be method templates.*

Consider the original `Grid` template with only one template parameter: the element type. You can instantiate grids of many different types, such as `ints` and `doubles`:

```
Grid<int> myIntGrid;
Grid<double> myDoubleGrid;
```

However, `Grid<int>` and `Grid<double>` are two different types. If you write a function that takes an object of type `Grid<double>`, you cannot pass a `Grid<int>`. Even though you know that the elements of an `int` grid could be copied to the elements of a `double` grid, because the `ints` could be coerced into `doubles`, you cannot assign an object of type `Grid<int>` to one of type `Grid<double>` or construct a `Grid<double>` from a `Grid<int>`. Neither of the following two lines compiles:

```
myDoubleGrid = myIntGrid; // DOES NOT COMPILE
Grid<double> newDoubleGrid(myIntGrid); // DOES NOT COMPILE
```

The problem is that the compiler-generated copy constructor and assignment operator for the `Grid` template are as follows:

```
Grid(const Grid<T>& src);
Grid<T>& operator=(const Grid<T>& rhs);
```

The `Grid` copy constructor and `operator=` both take a reference to a `const Grid<T>`. When you instantiate a `Grid<double>` and try to call the copy constructor and `operator=`, the compiler generates methods with these prototypes:

```
Grid(const Grid<double>& src);
Grid<double>& operator=(const Grid<double>& rhs);
```

Note that there are no constructors or `operator=` that take a `Grid<int>` within the generated `Grid<double>` class. However, you can rectify this oversight by adding templatized versions of the copy constructor and assignment operator to the `Grid` class to generate routines that will convert from one grid type to another. Here is the new `Grid` class definition:

```

template <typename T>
class Grid
{
public:
    explicit Grid(size_t inWidth = kDefaultWidth,
                  size_t inHeight = kDefaultHeight);
    virtual ~Grid();

    template <typename E>
    Grid(const Grid<E>& src);

    template <typename E>
    Grid<T>& operator=(const Grid<E>& rhs);

    // Omitted for brevity

private:
    template <typename E>
    void copyFrom(const Grid<E>& src);

    void initializeCellsContainer();
    std::vector<std::vector<T>> mCells;
    size_t mWidth, mHeight;
};

```

Examine the new templatized copy constructor first:

```

template <typename E>
Grid(const Grid<E>& src);

```

You can see that there is another template declaration with a different typename, *E* (short for “element”). The class is templatized on one type, *T*, and the new copy constructor is also templatized on a different type, *E*. This twofold templatization allows you to copy grids of one type to another. Here is the definition of the new copy constructor:

```

template <typename T>
template <typename E>
Grid<T>::Grid(const Grid<E>& src)
{
    copyFrom(src);
}

```

As you can see, you must declare the class template line (with the *T* parameter) before the member template line (with the *E* parameter). You can't combine them like this:

```

template <typename T, typename E> // Incorrect for nested template constructor!
Grid<T>::Grid(const Grid<E>& src)

```

This copy constructor uses the private and templatized `copyFrom()` method:

```

template <typename T>
template <typename E>
void Grid<T>::copyFrom(const Grid<E>& src)
{
    mWidth = src.getWidth();
    mHeight = src.getHeight();
    initializeCellsContainer();
}

```

```

        for (size_t i = 0; i < mWidth; i++) {
            for (size_t j = 0; j < mHeight; j++) {
                mCells[i][j] = src.getElementAt(i, j);
            }
        }
    }
}

```

In addition to the extra template parameter line before the `copyFrom()` method definition, note that you must use public accessor methods `getWidth()`, `getHeight()`, and `getElementAt()` to access the elements of `src`. That's because the object you're copying to is of type `Grid<T>`, and the object you're copying from is of type `Grid<E>`. They will not be the same type, so you must use public methods.

The final templated method is the assignment operator. Note that it takes a `const Grid<E>&` but returns a `Grid<T>&`:

```

template <typename T>
template <typename E>
Grid<T>& Grid<T>::operator=(const Grid<E>& rhs)
{
    copyFrom(rhs);
    return *this;
}

```

You need not check for self-assignment in the templated assignment operator, because assignment of the same types still happens in the old, non-templated, compiler-generated version of `operator=`, so there's no way you can get self-assignment here.

Method Templates with Non-Type Parameters

In the earlier example with integer template parameters for `HEIGHT` and `WIDTH`, you see that a major problem is that the height and width become part of the types. This restriction prevents you from assigning a grid with one height and width to a grid with a different height and width. In some cases, however, it's desirable to assign or copy a grid of one size to a grid of a different size. Instead of making the destination object a perfect clone of the source object, you would copy only those elements from the source array that fit in the destination array, padding the destination array with default values if the source array is smaller in either dimension. With method templates for the assignment operator and copy constructor, you can do exactly that, thus allowing assignment and copying of different sized grids. Here is the class definition:

```

template <typename T, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid
{
public:
    Grid() : mCells() {} // Zero-initialize mCells

    template <typename E, size_t WIDTH2, size_t HEIGHT2>
    Grid(const Grid<E, WIDTH2, HEIGHT2>& src);

    template <typename E, size_t WIDTH2, size_t HEIGHT2>
    Grid<T, WIDTH, HEIGHT>& operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs);

    void setElementAt(size_t x, size_t y, const T& inElem);
}

```

```

    T& getElementAt(size_t x, size_t y);
    const T& getElementAt(size_t x, size_t y) const;

    size_t getHeight() const { return HEIGHT; }
    size_t getWidth() const { return WIDTH; }

private:
    template <typename E, size_t WIDTH2, size_t HEIGHT2>
    void copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src);

    T mCells[WIDTH][HEIGHT];
};

```

This new definition includes method templates for the copy constructor and assignment operator, plus a helper method `copyFrom()`. When you write a copy constructor, the compiler stops generating a default constructor for you (details can be found in Chapter 7), so you have to add a default constructor as well. Note, however, that you do not need to write non-templatized copy constructor and assignment operator methods because the compiler-generated ones continue to be generated. They simply copy or assign `mCells` from the source to the destination, which is exactly the semantics you want for two grids of the same size.

When you templatize the copy constructor, assignment operator, and `copyFrom()`, you must specify all three template parameters. Here is the templatized copy constructor:

```

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid<T, WIDTH, HEIGHT>::Grid(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    copyFrom(src);
}

```

Here are the implementations of `copyFrom()` and `operator=`. Note that `copyFrom()` copies only `WIDTH` and `HEIGHT` elements in the `x` and `y` dimensions, respectively, from `src`, even if `src` is bigger than that. If `src` is smaller in either dimension, `copyFrom()` pads the extra spots with zero-initialized values. `T()` calls the default constructor for the object if `T` is a class type, or generates 0 if `T` is a simple type. This syntax is called the *zero-initialization* syntax. It's a good way to provide a reasonable default value for a variable whose type you don't yet know:

```

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>
void Grid<T, WIDTH, HEIGHT>::copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    for (size_t i = 0; i < WIDTH; i++) {
        for (size_t j = 0; j < HEIGHT; j++) {
            if (i < WIDTH2 && j < HEIGHT2) {
                mCells[i][j] = src.getElementAt(i, j);
            } else {
                mCells[i][j] = T();
            }
        }
    }
}
template <typename T, size_t WIDTH, size_t HEIGHT>

```

```
template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid<T, WIDTH, HEIGHT>& Grid<T, WIDTH, HEIGHT>::operator=(  
    const Grid<E, WIDTH2, HEIGHT2>& rhs)  
{  
    // No need to check for self-assignment because this version of  
    // assignment is never called when T and E are the same  
    copyFrom(rhs);  
    return *this;  
}
```

Class Template Specialization

You can provide alternate implementations of class templates for specific types. For example, you might decide that the `Grid` behavior for `const char*` (C-style strings) doesn't make sense. A `Grid<const char*>` will store its elements in a `vector<vector<const char*>>`. The copy constructor and assignment operator will perform shallow copies of this `const char*` pointer type. For `const char*`'s, it might make sense to do a deep copy of the string. The easiest solution for this is to write an alternative implementation specifically for `const char*`s, which stores the strings in a `vector<vector<string>>` and converts C-style strings into C++ strings so that their memory is automatically handled.

Alternate implementations of templates are called *template specializations*. You might find the syntax to be a little weird. When you write a class template specialization, you must specify that it's a template, and that you are writing the version of the template for that particular type. Here is the syntax for specializing the original version of the `Grid` for `const char*`s:

```
// When the template specialization is used, the original template must be  
// visible too. Including it here ensures that it will always be visible  
// when this specialization is visible.  
#include "Grid.h"  
  
template <>  
class Grid<const char*>  
{  
public:  
    explicit Grid(size_t inWidth = kDefaultWidth,  
                 size_t inHeight = kDefaultHeight);  
    virtual ~Grid();  
  
    void setElementAt(size_t x, size_t y, const char* inElem);  
    const char* getElementAt(size_t x, size_t y) const;  
  
    size_t getHeight() const { return mHeight; }  
    size_t getWidth() const { return mWidth; }  
    static const size_t kDefaultWidth = 10;  
    static const size_t kDefaultHeight = 10;  
  
private:  
    void initializeCellsContainer();  
    std::vector<std::vector<std::string>> mCells;  
    size_t mWidth, mHeight;  
};
```

Note that you don't refer to any type variable, such as `T`, in the specialization: you work directly with `const char*`s. One obvious question at this point is why this class is still a template. That is, what good is the following syntax?

```
template <>
class Grid<const char*>
```

This syntax tells the compiler that this class is a `const char*` specialization of the `Grid` class. Suppose that you didn't use that syntax and just tried to write this:

```
class Grid
```

The compiler wouldn't let you do that because there is already a class named `Grid` (the original class template). Only by specializing it can you reuse the name. The main benefit of specializations is that they can be invisible to the user. When a user creates a `Grid` of `ints` or `SpreadsheetCells`, the compiler generates code from the original `Grid` template. When the user creates a `Grid` of `const char*`s, the compiler uses the `const char*` specialization. This can all be "behind the scenes."

```
Grid<int> myIntGrid; // Uses original Grid template
Grid<const char*> stringGrid1(2, 2); // Uses const char* specialization

const char* dummy = "dummy";
stringGrid1.setElementAt(0, 0, "hello");
stringGrid1.setElementAt(0, 1, dummy);
stringGrid1.setElementAt(1, 0, dummy);
stringGrid1.setElementAt(1, 1, "there");

Grid<const char*> stringGrid2(stringGrid1);
```

When you specialize a template, you don't "inherit" any code: Specializations are not like derivations. You must rewrite the entire implementation of the class. There is no requirement that you provide methods with the same names or behavior. As an example, the `const char*` specialization of `Grid` implements only the `const` version of the `getElementAt()` method and omits the non-`const` version. As a matter of fact, you could write a completely different class with no relation to the original. Of course, that would abuse the template specialization ability, and you shouldn't do it without good reason. Here are the implementations for the methods of the `const char*` specialization. Unlike in the template definitions, you do not repeat the `template<>` syntax before each method or static member definition:

```
Grid<const char*>::Grid(size_t inWidth, size_t inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    initializeCellsContainer();
}

Grid<const char*>::~Grid()
{
    // Nothing to do, the vector will clean up itself.
}

void Grid<const char*>::initializeCellsContainer()
{
    mCells.resize(mWidth);
    for (auto& column : mCells) {
```

```

        column.resize(mHeight);
    }
}

void Grid<const char*>::setElementAt(size_t x, size_t y, const char* inElem)
{
    // Convert the given char* string into an std::string
    if (inElem)
        mCells[x][y] = inElem;
    else
        mCells[x][y] = "";
}

const char* Grid<const char*>::getElementAt(size_t x, size_t y) const
{
    return mCells[x][y].c_str();
}

```

This section discussed how to use class template specialization. It allows you to write a special implementation for a template with the template types replaced by specific types. Chapter 21 continues the discussion of specialization with a more advanced feature called *partial specialization*.

Deriving from Class Templates

You can inherit from class templates. If the derived class inherits from the template itself, it must be a template as well. Alternatively, you can derive from a specific instantiation of the class template, in which case your derived class does not need to be a template. As an example of the former, suppose you decide that the generic `Grid` class doesn't provide enough functionality to use as a game board. Specifically, you would like to add a `move()` method to the game board that moves a piece from one location on the board to another. Here is the class definition for the `GameBoard` template:

```

template <typename T>
class GameBoard : public Grid<T>
{
public:
    explicit GameBoard(size_t inWidth = Grid<T>::kDefaultWidth,
                       size_t inHeight = Grid<T>::kDefaultHeight);
    void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest);
};

```

This `GameBoard` template derives from the `Grid` template, and thereby inherits all its functionality. You don't need to rewrite `setElementAt()`, `getElementAt()`, or any of the other methods. You also don't need to add a copy constructor, `operator=`, or destructor, because you don't have any dynamically allocated memory in the `GameBoard`.

The inheritance syntax looks normal, except that the base class is `Grid<T>`, not `Grid`. The reason for this syntax is that the `GameBoard` template doesn't really derive from the generic `Grid` template. Rather, each instantiation of the `GameBoard` template for a specific type derives from the `Grid` instantiation for that type. For example, if you instantiate a `GameBoard` with a `ChessPiece` type, then the compiler generates code for a `Grid<ChessPiece>` as well. The "`: public Grid<T>`" syntax says that this class inherits from whatever `Grid` instantiation makes sense for the `T` type parameter. Note that the C++ name lookup rules for template inheritance require you to specify that `kDefaultWidth` and `kDefaultHeight` are declared in, and thus dependent on, the `Grid<T>` base class.

Here are the implementations of the constructor and the `move()` method. Again, note the use of `Grid<T>` in the call to the base class constructor. Additionally, although many compilers don't enforce it, the name lookup rules require you to use the `this` pointer to refer to data members and methods in the base class template:

```
template <typename T>
GameBoard<T>::GameBoard(size_t inWidth, size_t inHeight) :
    Grid<T>(inWidth, inHeight)
{
}
template <typename T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest)
{
    this->setElementAt(xDest, yDest, this->getElementAt(xSrc, ySrc));
    this->setElementAt(xSrc, ySrc, T()); // default construct the src cell
}
```

As you can see, `move()` uses the syntax `T()` described in the section on “Method Templates with Non-Type Parameters.”

You can use the `GameBoard` template as follows:

```
GameBoard<ChessPiece> chessboard(8, 8);
ChessPiece pawn;
chessBoard.setElementAt(0, 0, pawn);
chessBoard.move(0, 0, 0, 1);
```

Inheritance versus Specialization

Some programmers find the distinction between template inheritance and template specialization confusing. The following table summarizes the differences:

	INHERITANCE	SPECIALIZATION
Reuses code?	Yes: Derived classes contain all base class data members and methods.	No: you must rewrite all required code in the specialization.
Reuses name?	No: the derived class name must be different from the base class name.	Yes: the specialization must have the same name as the original.
Supports polymorphism?	Yes: objects of the derived class can stand in for objects of the base class.	No: each instantiation of a template for a type is a different type.

NOTE *Use inheritance for extending implementations and for polymorphism. Use specialization for customizing implementations for particular types.*

Alias Templates

Chapter 10 introduces the concept of a `typedef`. It allows you to give another name to specific types. To refresh your memory, you could, for example, write the following `typedef` to give a second name to type `int`:

```
typedef int MyInt;
```

Similarly, you can use a `typedef` to give another name to a templatized class. However, C++ requires you to specify concrete arguments for each template parameter. An example will make this clear. Suppose you have the following class template:

```
template<typename T1, typename T2>
class MyTemplateClass {/* ... */};
```

If you want to use a `typedef` to define another name for `MyTemplateClass`, you have to give concrete types for `T1` and `T2`. For example:

```
typedef MyTemplateClass<int, double> OtherName;
```

Specifying only one of the types, like the following example, is not valid in C++:

```
template<typename T1>
typedef MyTemplateClass<T1, double> OtherName; // Error
```

If you want to do something like this, you should use an *alias template*:

```
template<typename T1>
using OtherName = MyTemplateClass<T1, double>;
```

Pay special attention to the syntax. The new type name `OtherName` must be at the beginning with the alias template syntax, while it must be at the end for the `typedef` syntax.

Alternative Function Syntax

The *alternative function syntax* is introduced in Chapter 1, and is mentioned here again because it is a very useful feature in combination with templates. The problem solved with the alternative function syntax is that you don't always know the exact return type at the beginning of your function prototype. Take the following templatized function as an example:

```
template<typename Type1, typename Type2>
RetType myFunc(const Type1& t1, const Type2& t2) {return t1 + t2;}
```

In this example, `RetType` should be the type of the expression `t1+t2`, which isn't known yet at the beginning of the prototype line. `t1` and `t2` become known once the semantic analyzer reaches the end of the parameter list. Chapter 1 introduces the `decltype` feature, where `decltype(T)` returns the type of its argument `T`. With this knowledge, you might try to solve the previous return type issue by using the `decltype` feature as follows:

```
template<typename Type1, typename Type2>
decltype(t1+t2) myFunc(const Type1& t1, const Type2& t2) {return t1 + t2;}
```

Unfortunately, this is also not valid because `t1` and `t2` are still not yet defined when the compiler is parsing `decltype(t1+t2)`.

This problem is solved with the alternative function syntax. Note that in the new syntax, the return type is specified after the parameter list (trailing return type), hence the names of the parameters (and their types, and consequently the type `t1+t2`) are known:

```
template<typename Type1, typename Type2>
auto myFunc(const Type1& t1, const Type2& t2) -> decltype(t1+t2)
{
    return t1 + t2;
}
```



C++14 supports automatic function return-type deduction, which allows you to simplify `myFunc()` as follows. Note that this uses the `auto` keyword but omits the trailing return type.

```
template<typename Type1, typename Type2>
auto myFunc(const Type1& t1, const Type2& t2)
{
    return t1 + t2;
}
```

FUNCTION TEMPLATES

You can also write templates for stand-alone functions. For example, you could write a generic function to find a value in an array and return its index:

```
static const size_t NOT_FOUND = (size_t)(-1);
template <typename T>
size_t Find(T& value, T* arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        if (arr[i] == value) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND
}
```

The `Find()` function template can work on arrays of any type. For example, you could use it to find the index of an `int` in an array of `ints` or a `SpreadsheetCell` in an array of `SpreadsheetCells`.

You can call the function in two ways: explicitly specifying the type with angle brackets or omitting the type and letting the compiler *deduce* it from the arguments. Here are some examples:

```
int x = 3, intArr[] = {1, 2, 3, 4};
size_t sizeIntArr = sizeof(intArr) / sizeof(int);
size_t res;
res = Find(x, intArr, sizeIntArr);           // calls Find<int> by deduction
res = Find<int>(x, intArr, sizeIntArr); // calls Find<int> explicitly
if (res != NOT_FOUND)
    cout << res << endl;
else
    cout << "Not found" << endl;

double d1 = 5.6, dArr[] = {1.2, 3.4, 5.7, 7.5};
size_t sizeDoubleArr = sizeof(dArr) / sizeof(double);
res = Find(d1, dArr, sizeDoubleArr);           // calls Find<double> by deduction
```

```

res = Find<double>(d1, dArr, sizeDoubleArr); // calls Find<double> explicitly
if (res != NOT_FOUND)
    cout << res << endl;
else
    cout << "Not found" << endl;

//res = Find(x, dArr, sizeDoubleArr); // DOES NOT COMPILE!
// Arguments are different types.

SpreadsheetCell c1(10), c2Arr[2] =
    {SpreadsheetCell(4), SpreadsheetCell(10)};
size_t sizeC2Arr = sizeof(c2Arr) / sizeof(SpreadsheetCell);
res = Find(c1, c2Arr, sizeC2Arr);
res = Find<SpreadsheetCell>(c1, c2Arr, sizeC2Arr);

```

The previous implementation of the `Find()` function requires the size of the array as one of the parameters. Sometimes the compiler knows the exact size of an array; for example, for stack-based arrays. It would be nice to be able to call the `Find()` function with such arrays without the need to pass it the size of the array. This can be accomplished by adding the following function template. The implementation just forwards the call to the previous `Find()` function template. This also demonstrates that function templates can take non-type parameters, just as class templates.

```

template <typename T, size_t S>
size_t Find(T& value, T(&arr) [S])
{
    return Find(value, arr, S);
}

```

This version of `Find()` can be called as follows:

```

int x = 3, intArr[] = {1, 2, 3, 4};
size_t res = Find(x, intArr);

```

Like class template method definitions, function template definitions (not just the prototypes) must be available to all source files that use them. Thus, you should put the definitions in header files if more than one source file uses them.

Template parameters of function templates can have defaults, just as class templates.

NOTE *The C++ standard library provides a templatized `std::find()` function that is more powerful than the one above. See Chapter 17 for details.*

Function Template Specialization

Just as you can specialize class templates, you can specialize function templates. For example, you might want to write a `Find()` function for `const char*` C-style strings that compares them with `strcmp()` instead of `operator==`. Here is a specialization of the `Find()` function to do this:

```

template<>
size_t Find<const char*>(const char*& value, const char** arr, size_t size)
{
    cout << "Specialization" << endl;

```

```

        for (size_t i = 0; i < size; i++) {
            if (strcmp(arr[i], value) == 0) {
                return i; // Found it; return the index
            }
        }
        return NOT_FOUND; // Failed to find it; return NOT_FOUND
    }
}

```

You can omit the `<const char*>` in the function name when the parameter type can be deduced from the arguments, making your prototype look like this:

```

template<>
size_t Find(const char*& value, const char** arr, size_t size)

```

However, the deduction rules are tricky when you involve overloading as well (see the next section), so, in order to avoid mistakes, it's better to note the type explicitly.

Although the specialized `find()` function could take just `const char*` instead of `const char*&` as its first parameter, it's best to keep the arguments parallel to the non-specialized version of the function for the deduction rules to function properly.

You can use the specialization as follows:

```

const char* word = "two";
const char* arr[] = {"one", "two", "three", "four"};
size_t sizeArr = sizeof(arr) / sizeof(arr[0]);
size_t res;
res = Find<const char*>(word, arr, sizeArr); // Calls const char* specialization
res = Find(word, arr, sizeArr); // Calls const char* specialization

```

Function Template Overloading

You can also overload function templates with non-template functions. For example, instead of writing a `Find()` template specialization for `const char*`, you could write a non-template `Find()` function that works on `const char*`s:

```

size_t Find(const char*& value, const char** arr, size_t size)
{
    cout << "overload" << endl;
    for (size_t i = 0; i < size; i++) {
        if (strcmp(arr[i], value) == 0) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND
}

```

This function is identical in behavior to the specialized version in the previous section. However, the rules for when it is called are different:

```

const char* word = "two";
const char* arr[] = {"one", "two", "three", "four"};
size_t sizeArr = sizeof(arr) / sizeof(arr[0]);
size_t res;
res = Find<const char*>(word, arr, sizeArr); // Calls template with T=const char*
res = Find(word, arr, sizeArr); // Calls non-template function!

```

Thus, if you want your function to work both when `const char*` is explicitly specified and via deduction when it is not, you should write a specialized template version instead of a non-template, overloaded version.

Function Template Overloading and Specialization Together

It's possible to write both a specialized `Find()` template for `const char*`s and a stand-alone `Find()` function for `const char*`s. The compiler always prefers the non-template function to a templatized version. However, if you specify the template instantiation explicitly, the compiler will be forced to use the template version:

```
const char* word = "two";
const char* arr[] = {"one", "two", "three", "four"};
size_t sizeArr = sizeof(arr) / sizeof(arr[0]);
size_t res;
res = Find<const char*>(word, arr, sizeArr); // Calls const char* specialization
                                                // of the template
res = Find(word, arr, sizeArr);               // Calls the Find non-template function
```

Friend Function Templates of Class Templates

Function templates are useful when you want to overload operators in a class template. For example, you might want to overload the addition operator (`operator+`) for the `Grid` class template to be able to add two grids together. The result will be a `Grid` with the same size as the smallest `Grid` of the two operands. Suppose you want to make your `operator+` a stand-alone function template. The definition, which should go directly in `Grid.h`, looks as follows:

```
template <typename T>
Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs)
{
    size_t minWidth = std::min(lhs.getWidth(), rhs.getWidth());
    size_t minHeight = std::min(lhs.getHeight(), rhs.getHeight());

    Grid<T> result(minWidth, minHeight);
    for (size_t y = 0; y < minHeight; ++y) {
        for (size_t x = 0; x < minWidth; ++x) {
            result.setElementAt(x, y, lhs.mCells[x][y] + rhs.mCells[x][y]);
        }
    }
    return result;
}
```

This function template will work on any `Grid`, as long as there is an addition operator for the elements of the grid. The only problem with this implementation is that it accesses private members of the `Grid` class. The obvious solution is to use the public `getElementAt()` method, but let's see how you can make a function template a friend of a class template. For this example, you can make the operator a friend of the `Grid` class. However, both the `Grid` class and the `operator+` are templates. What you really want is for each instantiation of `operator+` for a particular type `T` to be a friend of the `Grid` template instantiation for that type. The syntax looks like this:

```

// Forward declare Grid template.
template <typename T> class Grid;

// Prototype for templatized operator+.
template<typename T>
Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs);

template <typename T>
class Grid
{
public:
    // Omitted for brevity
    friend Grid<T> operator+ <T>(const Grid<T>& lhs, const Grid<T>& rhs);
    // Omitted for brevity
};

```

This friend declaration is tricky: you're saying that, for an instance of the template with type `T`, the `T` instantiation of `operator+` is a `friend`. In other words, there is a one-to-one mapping of friends between the class instantiations and the function instantiations. Note particularly the explicit template specification `<T>` on `operator+` (the space after `operator+` is optional, but in the interest of readability it should always be there). This syntax tells the compiler that `operator+` is itself a template.

C++14

VARIABLE TEMPLATES

In addition to class templates, class method templates, and function templates, C++14 adds the ability to write *variable templates*. The syntax is as follows:

```

template <typename T>
constexpr T pi = T(3.1415926535897932385);

```

This is a variable template for the value of PI. To get PI in a certain type you use the following syntax:

```

float piInt = pi<float>;
long double piLongDouble = pi<long double>;

```

You will always get the closest value of PI representable in the requested type. Just as other types of templates, variable templates can also be specialized.

SUMMARY

This chapter started a discussion on using templates for generic programming. You saw the syntax on how to write templates and examples where templates are really useful. It explained how to write class templates, how to organize your code in different files, how to use template parameters, and how to templatize methods of a class. It further discussed how to use class template specialization to write special implementations of a template where the template parameters are replaced with specific arguments. The chapter finished with an explanation of function templates and variable templates.

Chapter 21 continues the discussion on templates with some more advanced features such as variadic templates and metaprogramming.

12

Demystifying C++ I/O

WHAT'S IN THIS CHAPTER?

- What streams are
- How to use streams for input and output of data
- What the available standard streams are in the Standard Library

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

A program's fundamental job is to accept input and produce output. A program that produces no output of any sort would not be very useful. All languages provide some mechanism for I/O, either as a built-in part of the language or through an OS-specific API. A good I/O system is both flexible and easy to use. Flexible I/O systems support input and output through a variety of devices, such as files and the user console. They also support reading and writing of different types of data. I/O is error-prone because data coming from a user can be incorrect or the underlying file system or other data source can be inaccessible. Thus, a good I/O system is also capable of handling error conditions.

If you are familiar with the C language, you have undoubtedly used `printf()` and `scanf()`. As I/O mechanisms, `printf()` and `scanf()` are certainly flexible. Through escape codes and variable placeholders, they can be customized to read in specially formatted data, or output any value that the formatting codes permit, which is currently limited to integer/character values, floating point values, and strings. However, `printf()` and `scanf()` falter on other measures of good I/O systems. They do not handle errors particularly well, they are not

flexible enough to handle custom data types, and in an object-oriented language like C++, they are not at all object oriented.

C++ provides a more refined method of input and output through a mechanism known as *streams*. Streams are a flexible and object-oriented approach to I/O. In this chapter, you will learn how to use streams for data output and input. You will also learn how to use the stream mechanism to read from various sources and write to various destinations, such as the user console, files, and even strings. This chapter covers the most commonly used I/O features.

USING STREAMS

The stream metaphor takes a bit of getting used to. At first, streams may seem more complex than traditional C-style I/O, such as `printf()`. In reality, they seem complicated initially only because there is a deeper metaphor behind streams than there is behind `printf()`. Don't worry though; after a few examples, you'll never look back.

What Is a Stream, Anyway?

Chapter 1 compares the `cout` stream like a laundry chute for data. You throw some variables down the stream, and they are written to the user's screen, or *console*. More generally, all streams can be viewed as data chutes. Streams vary in their direction and their associated source or destination. For example, the `cout` stream that you are already familiar with is an output stream, so its direction is "out." It writes data to the console so its associated destination is "console." There is another standard stream called `cin` that accepts input from the user. Its direction is "in," and its associated source is "console." Both `cout` and `cin` are predefined instances of streams that are defined within the `std` namespace in C++. The following table gives a brief description of all predefined streams. The difference between *buffered* and *unbuffered* streams is explained in a later section:

STREAM	DESCRIPTION
<code>cin</code>	An input stream, reads data from the "input console."
<code>cout</code>	A buffered output stream, writes data to the "output console."
<code>cerr</code>	An unbuffered output stream, writes data to the "error console," which is often the same as the "output console."
<code>clog</code>	A buffered version of <code>cerr</code> .

Note that graphical user interface applications normally do not have a console; i.e., if you write something to `cout`, the user will not see it. If you are writing a library, you should never assume the existence of `cout`, `cin`, `cerr` or, `clog` because you never know if your library will be used in a console or in a GUI application.

NOTE Every input stream has an associated source. Every output stream has an associated destination.

Another important aspect of streams is that they include data but also have a so-called *current position*. The current position is the position in the stream where the next read or write operation will take place.

Stream Sources and Destinations

Streams as a concept can be applied to any object that accepts data or emits data. You could write a stream-based network class or stream-based access to a MIDI-based instrument. In C++, there are three common sources and destinations for streams.

You have already read many examples of user, or console, streams. Console input streams make programs interactive by allowing input from the user at run time. Console output streams provide feedback to the user and output results.

File streams, as the name implies, read data from a file system and write data to a file system. File input streams are useful for reading configuration data and saved files, or for batch processing file-based data. File output streams are useful for saving state and providing output. File streams subsume the functionality of the C functions `fprintf()`, `fwrite()`, and `fputs()` for output, and `fscanf()`, `fread()`, and `fgets()` for input.

String streams are an application of the stream metaphor to the string type. With a string stream, you can treat character data just as you would treat any other stream. For the most part, this is merely a handy syntax for functionality that could be handled through methods on the `string` class. However, using stream syntax provides opportunities for optimization and can be far more convenient than direct use of the `string` class. String streams subsume the functionality of `sprintf()`, `sprintf_s()`, `sscanf()`, and other forms of C string formatting functions.

The rest of this section deals with console streams (`cin` and `cout`). Examples of file and string streams are provided later in this chapter. Other types of streams, such as printer output or network I/O are often platform dependent, so they are not covered in this book.

Output with Streams

Output using streams is introduced in Chapter 1 and is used in almost every chapter in this book. This section briefly revisits some of the basics and introduces material that is more advanced.

Output Basics

Output streams are defined in the `<ostream>` header file. Most programmers include `<iostream>` in their programs, which in turn includes the headers for both input streams and output streams. The `<iostream>` header also declares the standard console output stream, `cout`.

The `<<` operator is the simplest way to use output streams. C++ basic types, such as ints, pointers, doubles, and characters, can be output using `<<`. In addition, the C++ `string` class is compatible with `<<`, and C-style strings are properly output as well. Following are some examples of using `<<`:

```
int i = 7;
cout << i << endl;

char ch = 'a';
cout << ch << endl;

string myString = "Hello World.";
cout << myString << endl;
```

The output is as follows:

```
7
a
Hello World.
```

The `cout` stream is the built-in stream for writing to the console, or *standard output*. You can “chain” uses of `<<` together to output multiple pieces of data. This is because the `<<` operator returns a reference to the stream as its result so you can immediately use `<<` again on the same stream. For example:

```
int j = 11;
cout << "The value of j is " << j << "!" << endl;
```

The output is as follows:

```
The value of j is 11!
```

C++ streams correctly parse C-style escape codes, such as strings that contain `\n`. You can also use `std::endl` to start a new line. The difference between using `\n` and `endl` is that `\n` just starts a new line while `endl` also flushes the buffer. Watch out with `endl` because too many flushes might hurt performance. The following example uses `endl` to output and flush several lines of text with just one line of code.

```
cout << "Line 1" << endl << "Line 2" << endl << "Line 3" << endl;
```

The output is as follows:

```
Line 1
Line 2
Line 3
```

Methods of Output Streams

The `<<` operator is, without a doubt, the most useful part of output streams. However, there is additional functionality to be explored. If you take a peek at the `<ostream>` header file, you’ll see many lines of overloaded definitions of the `<<` operator. You’ll also find some useful public methods.

put() and write()

`put()` and `write()` are *raw output methods*. Instead of taking an object or variable that has some defined behavior for output, `put()` accepts a single character, while `write()` accepts a character array. The data passed to these methods is output as is, without any special formatting or processing. For example, the following function takes a C-style string and outputs it to the console without using the `<<` operator:

```
void rawWrite(const char* data, int dataSize)
{
    cout.write(data, dataSize);
}
```

The next function writes the given index of a C-style string to the console by using the `put()` method:

```
void rawPutChar(const char* data, int charIndex)
{
    cout.put(data[charIndex]);
}
```

flush()

When you write to an output stream, the stream does not necessarily write the data to its destination right away. Most output streams *buffer*, or accumulate data instead of writing it out as it comes in. The stream will *flush*, or write out the accumulated data, when one of the following conditions occurs:

- A sentinel, such as the `endl` marker, is reached.
- The stream goes out of scope and is destructed.
- Input is requested from a corresponding input stream (i.e., when you make use of `cin` for input, `cout` will flush). In the section on file streams, you learn how to establish this type of link.
- The stream buffer is full.
- You explicitly tell the stream to flush its buffer.

One way to explicitly tell a stream to flush is to call its `flush()` method, as in the code that follows:

```
cout << "abc";
cout.flush();    // abc is written to the console.
cout << "def";
cout << endl;    // def is written to the console.
```

NOTE Not all output streams are buffered. The `cerr` stream, for example, does not buffer its output.

Handling Output Errors

Output errors can arise in a variety of situations. Perhaps you are trying to open a non-existing file. Maybe a disk error has prevented a write operation from succeeding, for example because the disk is full. None of the streams' code you have read up until this point has considered these possibilities, mainly for brevity. However, it is vital that you address any error conditions that occur.

When a stream is in its normal usable state, it is said to be “good.” The `good()` method can be called directly on a stream to determine whether or not the stream is currently good.

```
if (cout.good()) {
    cout << "All good" << endl;
}
```

The `good()` method provides an easy way to obtain basic information about the validity of the stream, but it does not tell you why the stream is unusable. There is a method called `bad()` that provides a bit more information. If `bad()` returns `true`, it means that a fatal error has occurred (as opposed to any nonfatal condition like end-of-file). Another method, `fail()`, returns `true` if the most recent operation has failed; however, it doesn't say anything about the next operation which can either succeed or fail as well. For example, after calling `flush()` on an output stream, you could call `fail()` to make sure the flush was successful.

```
cout.flush();
if (cout.fail()) {
    cerr << "Unable to flush to standard out" << endl;
}
```

You can also tell the streams to throw exceptions when a failure occurs. You then write a `catch` handler to catch `ios_base::failure` exceptions on which you can use the `what()` method to get a description of the error and the `code()` method to get the error code. However, whether or not you get useful information depends on the STL implementation that you use.

```
cout.exceptions(ios::failbit | ios::badbit | ios::eofbit);
try {
    cout << "Hello World." << endl;
} catch (const ios_base::failure& ex) {
    cerr << "Caught exception: " << ex.what()
        << ", error code = " << ex.code() << endl;
}
```

To reset the error state of a stream, use the `clear()` method:

```
cout.clear();
```

Error checking is performed less frequently for console output streams than for file output streams or input streams. The methods discussed here apply for other types of streams as well and are revisited later as each type is discussed.

Output Manipulators

One of the unusual features of streams is that you can throw more than just data down the chute. C++ streams also recognize *manipulators*, objects that make a change to the behavior of the stream instead of, or in addition to, providing data for the stream to work with.

You have already seen one manipulator: `endl`. The `endl` manipulator encapsulates data and behavior. It tells the stream to output an end-of-line sequence and to flush its buffer. Following are some other useful manipulators, many of which are defined in the `<iostream>` and `<iomanip>` standard header files. The example after this list shows how to use them:

- **boolalpha** and **noboolalpha**. Tells the stream to output `bool` values as *true* and *false* (`boolalpha`) or 1 and 0 (`noboolalpha`). The default is `noboolalpha`.
- **hex**, **oct**, and **dec**. Outputs numbers in hexadecimal, octal, and base 10, respectively.
- **setprecision**. Sets the number of decimal places that are output for fractional numbers. This is a parameterized manipulator (meaning that it takes an argument).
- **setw**. Sets the field width for outputting numerical data. This is a parameterized manipulator.
- **setfill**. Specifies the character that is used to pad numbers that are smaller than the specified width. This is a parameterized manipulator.
- **showpoint** and **noshowpoint**. Forces the stream to always or never show the decimal point for floating point numbers with no fractional part.
- **put_money**. Writes a formatted money amount to a stream.
- **put_time**. Writes a formatted time to a stream.
- **quoted**. A parameterized manipulator that encloses a given string with quotes and escapes embedded quotes.

C++14

All the above manipulators stay in effect for subsequent output to the stream until they are reset, except `setw` which is only active for the next single output. The following example uses several of these manipulators to customize its output.

```
// Boolean values
bool myBool = true;
cout << "This is the default: " << myBool << endl;
cout << "This should be true: " << boolalpha << myBool << endl;
cout << "This should be 1: " << noboolalpha << myBool << endl;

// Simulate "%6d" with streams
int i = 123;
printf("This should be ' 123': %6d\n", i);
cout << "This should be ' 123': " << setw(6) << i << endl;
// Simulate "%06d" with streams
printf("This should be '000123': %06d\n", i);
cout << "This should be '000123': " << setfill('0') << setw(6) << i << endl;

// Fill with *
cout << "This should be '***123': " << setfill('*') << setw(6) << i << endl;
// Reset fill character
```

```

cout << setfill(' ');
// Floating point values
double dbl = 1.452;
double dbl2 = 5;
cout << "This should be ' 5': " << setw(2) << noshowpoint << dbl2 << endl;
cout << "This should be @@1.452: " << setw(7) << setfill('@') << dbl << endl;
// Reset fill character
cout << setfill(' ');

// Instructs cout to start formatting numbers according to your location.
// Chapter 18 explains the details of the imbuf call and the locale object.
cout.imbuf(locale(""));
// Format numbers according to your location
cout << "This is 1234567 formatted according to your location: " << 1234567 << endl;

// Money amount
cout << "This should be a money amount of 120000, "
    << "formatted according to your location: "
    << put_money("120000") << endl;

// Date and time
time_t t_t = time(nullptr); // Get current system time
tm* t = localtime(&t_t); // Convert to local time
cout << "This should be the current date and time "
    << "formatted according to your location: "
    << put_time(t, "%c") << endl;

// C++14: Quoted string
cout << "This should be: \"Quoted string with \\\\"embedded quotes\\\\".\\\""
    << quoted("Quoted string with \"embedded quotes\".") << endl;

```

NOTE *This example might give you a security-related error or warning on the call to localtime(). With Microsoft Visual Studio you can use the safe version called localtime_s(). On Linux you can use localtime_r().*

If you don't care for the concept of manipulators, you can usually get by without them. Streams provide much of the same functionality through equivalent methods like `precision()`. For example, take the following line:

```
cout << "This should be '1.2346': " << setprecision(5) << 1.23456789 << endl;
```

This can be converted to use a method call as follows. The advantage of the method calls is that they return the previous value, allowing you to restore it.

```
cout.precision(5);
cout << "This should be '1.2346': " << 1.23456789 << endl;
```

For a detailed description of all stream methods and manipulators consult a Standard Library Reference, for example <http://www.cppreference.com> or <http://www.cplusplus.com/reference/>.

Input with Streams

Input streams provide a simple way to read in structured or unstructured data. In this section, the techniques for input are discussed within the context of `cin`, the console input stream.

Input Basics

There are two easy ways to read data by using an input stream. The first is an analog of the `<<` operator that outputs data to an output stream. The corresponding operator for reading data is `>>`. When you use `>>` to read data from an input stream, the variable you provide is the storage for the received value. For example, the following program reads one word from the user and puts it into a string. Then the string is output back to the console:

```
string userInput;
cin >> userInput;
cout << "User input was " << userInput << endl;
```

By default, the `>>` operator tokenizes values according to white space. For example, if a user runs the previous program and enters `hello` there as input, only the characters up to the first white space character (the space character in this instance) will be captured into the `userInput` variable. The output would be as follows:

```
User input was hello
```

One solution to include white space in the input is to use `get()`, discussed later in this chapter.

The `>>` operator works with different variable types, just like the `<<` operator. For example, to read an integer, the code differs only in the type of the variable:

```
int userInput;
cin >> userInput;
cout << "User input was " << userInput << endl;
```

You can use input streams to read in multiple values, mixing and matching types as necessary. For example, the following function, an excerpt from a restaurant reservation system, asks the user for a last name and the number of people in their party:

```
void getReservationData()
{
    string guestName;
    int partySize;
    cout << "Name and number of guests: ";
    cin >> guestName >> partySize;
    cout << "Thank you, " << guestName << "." << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

Remember that the `>>` operator tokenizes values according to white space, so the `getReservationData()` function does not allow you to enter a name with white space. A solution using `unget()` is discussed later in this chapter. Note also that even though the use of `cout` does not explicitly flush the buffer using `endl` or `flush()`, the text will still be written to the console because the use of `cin` immediately flushes the `cout` buffer; they are linked together in this way.

NOTE *If you get confused between `<<` and `>>`, just think of the angles as pointing toward their destination. In an output stream, `<<` points toward the stream itself because data is being sent to the stream. In an input stream, `>>` points toward the variables because data is being stored.*

Input Methods

Just like output streams, input streams have several methods that allow a lower level of access than the functionality provided by the more common `>>` operator.

`get()`

The `get()` method allows raw input of data from a stream. The simplest version of `get()` returns the next character in the stream, though other versions exist that read multiple characters at once. `get()` is most commonly used to avoid the automatic tokenization that occurs with the `>>` operator. For example, the following function reads a name, which can be made up of several words, from an input stream until the end of the stream is reached:

```
string readName(istream& inStream)
{
    string name;
    while (!inStream.fail()) {
        int next = inStream.get();
        if (next == std::char_traits<char>::eof())
            break;
        name += static_cast<char>(next); // Append character.
    }
    return name;
}
```

There are several interesting observations to make about this `readName()` function:

- Its parameter is a non-const reference to an `istream`, not a `const` reference. The methods that read in data from a stream will change the actual stream (most notably, its position), so they are not `const` methods. Thus, you can't call them on a `const` reference.
- The return value of `get()` is stored in an `int`, not in a `char`. Because `get()` can return special non-character values such as `std::char_traits<char>::eof()` (end-of-file), `ints` are used.

`readName()` is a bit strange because there are two ways to get out of the loop. Either the stream can get into a failed state, or the end of the stream is reached. A more common pattern for reading

from a stream uses a different version of `get()` that takes a reference to a character and returns a reference to the stream. This pattern takes advantage of the fact that evaluating an input stream within a conditional context results in `true` only if the stream is available for additional reading. Encountering an error or reaching the end-of-file both cause the stream to evaluate to `false`. The underlying details of the conversion operations required to implement this feature are explained in Chapter 14. The following version of the same function is a bit more concise:

```
string readName(istream& inStream)
{
    string name;
    char next;
    while (inStream.get(next)) {
        name += next;
    }
    return name;
}
```

unget()

For most purposes, the correct way to think of an input stream is as a one-way chute. Data falls down the chute and into variables. The `unget()` method breaks this model in a way by allowing you to push data back up the chute.

A call to `unget()` causes the stream to back up by one position, essentially putting the previous character read back on the stream. You can use the `fail()` method to see if `unget()` was successful or not. For example, `unget()` can fail if the current position is at the beginning of the stream.

The `getReservationData()` function seen earlier in this chapter did not allow you to enter a name with white space. The following code uses `unget()` to allow white space in the name. The code reads character by character and checks whether the character is a digit or not. If the character is not a digit, it is added to `guestName`. If it is a digit, the character is put back into the stream using `unget()`, the loop is stopped, and the `>>` operator is used to input an integer, `partySize`. The meaning of `noskipws` is discussed later in the section “Input Manipulators.”

```
void getReservationData()
{
    string guestName;
    int partySize = 0;
    // Read characters until we find a digit
    char ch;
    cin >> noskipws;
    while (cin >> ch) {
        if (isdigit(ch)) {
            cin.unget();
            if (cin.fail())
                cout << "unget() failed" << endl;
            break;
        }
        guestName += ch;
    }
    // Read partysize
```

```
    cin >> partySize;
    cout << "Thank you '" << guestName
        << "', party of " << partySize << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

putback()

`putback()`, like `unget()`, lets you move backward by one character in an input stream. The difference is that the `putback()` method takes the character being placed back on the stream as a parameter:

```
char ch1;
cin >> ch1;
cin.putback(ch1);
// ch1 will be the next character read off the stream.
```

peek()

The `peek()` method allows you to preview the next value that *would* be returned if you were to call `get()`. To take the chute metaphor perhaps a bit too far, you could think of it as looking up the chute without a value actually falling down it.

`peek()` is ideal for any situation where you need to look ahead before reading a value. For example, the following code implements the `getReservationData()` function that allows white space in the name, but uses `peek()` instead of `unget()`:

```
void getReservationData()
{
    string guestName;
    int partySize = 0;
    // Read characters until we find a digit
    char ch;
    cin >> noskipws;
    while (true) {
        // 'peek' at next character
        ch = static_cast<char>(cin.peek());
        if (!cin.good())
            break;
        if (isdigit(ch)) {
            // next character will be a digit, so stop the loop
            break;
        }
        // next character will be a non-digit, so read it
        cin >> ch;
        guestName += ch;
    }
    // Read partysize
    cin >> partySize;
    cout << "Thank you '" << guestName
        << "', party of " << partySize << endl;
    if (partySize > 10) {
```

```

        cout << "An extra gratuity will apply." << endl;
    }
}

```

getline()

Obtaining a single line of data from an input stream is so common that a method exists to do it for you. The `getline()` method fills a character buffer with a line of data up to the specified size. The specified size includes the `\0` character. Thus, the following code will read a maximum of `kBufferSize-1` characters from `cin`, or until an end-of-line sequence is read:

```

char buffer[kBufferSize];
cin.getline(buffer, kBufferSize);

```

When `getline()` is called, it reads a line from the input stream, up to and including the end-of-line sequence. However, the end-of-line character or characters do not appear in the string. Note that the end-of-line sequence is platform dependent. For example, it can be `\r\n`, or `\n`, or `\n\r`.

There is a form of `get()` that performs the same operation as `getline()`, except that it leaves the newline sequence in the input stream.

There is also a function called `getline()` that can be used with C++ strings. It is defined in the `<string>` header file and is in the `std` namespace. It takes a stream reference, a `string` reference, and an optional delimiter as parameters. The advantage of using this version of `getline()` is that it doesn't require you to specify the size of the buffer.

```

string myString;
std::getline(cin, myString);

```

Handling Input Errors

Input streams have a number of methods to detect unusual circumstances. Most of the error conditions related to input streams occur when there is no data available to read. For example, the end of stream (referred to as *end-of-file*, even for non-file streams) may have been reached. The most common way of querying the state of an input stream is to access it within a conditional. For example, the following loop will keep looping as long as `cin` remains in a good state:

```
while (cin) { ... }
```

You can input data at the same time:

```
while (cin >> ch) { ... }
```

You can also call the `good()`, `bad()`, and `fail()` methods, just like on output streams. There is also an `eof()` method that returns `true` if the stream has reached its end.

You should also get into the habit of checking the stream state after reading data so that you can recover from bad input.

The following program shows the common pattern for reading data from a stream and handling errors. The program reads numbers from standard input and displays their sum once end-of-file is reached. Note that in command-line environments, the end-of-file is indicated by the user typing a particular character. In Unix and Linux, it is Control+D, in Windows it is Control+Z. The exact character is operating system dependent, so you will need to know what your operating system requires:

```

cout << "Enter numbers on separate lines to add. "
      << "Use Control+D to finish (Control+Z in Windows)." << endl;
int sum = 0;
if (!cin.good()) {
    cerr << "Standard input is in a bad state!" << endl;
    return 1;
}
int number;
while (!cin.bad()) {
    cin >> number;
    if (cin.good()) {
        sum += number;
    } else if (cin.eof()) {
        break; // Reached end of file
    } else if (cin.fail()) {
        // Failure!
        cin.clear(); // Clear the failure state.
        string badToken;
        cin >> badToken; // Consume the bad input.
        cerr << "WARNING: Bad input encountered: " << badToken << endl;
    }
}
cout << "The sum is " << sum << endl;

```

Input Manipulators

The built-in input manipulators, described in the list that follows, can be sent to an input stream to customize the way that data is read.

- **boolalpha** and **noboolalpha**. If **boolalpha** is used, the string *false* will be interpreted as a Boolean value *false*; anything else will be treated as the Boolean value *true*. If **noboolalpha** is set, 0 will be interpreted as *false*, anything else as *true*. The default is **noboolalpha**.
- **hex**, **oct**, and **dec**. Reads numbers in hexadecimal, octal, and base 10, respectively.
- **skipws** and **noskipws**. Tells the stream to either skip white space when tokenizing or to read in white space as its own token.
- **ws**. A handy manipulator that simply skips over the current series of white space at the current position in the stream.
- **get_money**. Reads a money amount from a stream.
- **get_time**. Reads a formatted time from a stream.
- **quoted**. A parameterized manipulator that reads a string enclosed with quotes and in which embedded quotes are escaped.

Input is locale aware. For example, the following code enables your system locale for `cin`. Locales are discussed in Chapter 18:

```
cin.imbue(locale(""));
int i;
cin >> i;
```

If your system locale is U.S. English, you can enter `1,000` and it will be parsed as `1000`. On the other hand, if your system locale is Dutch Belgium, you should enter `1.000` to get the value of `1000`.

Input and Output with Objects

You can use the `<<` operator to output a C++ string even though it is not a basic type. In C++, objects are able to prescribe how they are output and input. This is accomplished by *overloading* the `<<` and `>>` operator to understand a new type or class.

Why would you want to overload these operators? If you are familiar with the `printf()` function in C, you know that it is not flexible in this area. `printf()` knows about several types of data, but there really isn't a way to give it additional knowledge. For example, consider the following simple class:

```
class Muffin
{
public:
    const string& getDescription() const;
    void setDescription(const string& inDesc);

    int getSize() const;
    void setSize(int inSize);

    bool getHasChocolateChips() const;
    void setHasChocolateChips(bool inChips);

private:
    string mDesc;
    int mSize;
    bool mHasChocolateChips;
};

const string& Muffin::getDescription() const { return mDesc; }
void Muffin::setDescription(const string& inDesc) { mDesc = inDesc; }
int Muffin::getSize() const { return mSize; }
void Muffin::setSize(int inSize) { mSize = inSize; }
bool Muffin::getHasChocolateChips() const { return mHasChocolateChips; }
void Muffin::setHasChocolateChips(bool inChips) { mHasChocolateChips = inChips; }
```

To output an object of class `Muffin` by using `printf()`, it would be nice if you could specify it as an argument, perhaps using `%m` as a placeholder:

```
printf("Muffin output: %m\n", myMuffin); // BUG! printf doesn't understand Muffin.
```

Unfortunately, the `printf()` function knows nothing about the `Muffin` type and is unable to output an object of type `Muffin`. Worse still, because of the way the `printf()` function is declared, this will result in a run-time error, not a compile-time error (though a good compiler will give you a warning).

The best you can do with `printf()` is to add a new `output()` method to the `Muffin` class:

```
class Muffin
{
public:
    const string& getDescription() const;
    void    setDescription(const string& inDesc);

    int    getSize() const;
    void    setSize(int inSize);

    bool   getHasChocolateChips() const;
    void    setHasChocolateChips(bool inChips);

    void    output();
private:
    string mDesc;
    int    mSize;
    bool   mHasChocolateChips;
};

// Other method implementations omitted for brevity
void Muffin::output()
{
    printf("%s, Size is %d, %s\n", getDescription().c_str(), getSize(),
           (getHasChocolateChips() ? "has chips" : "no chips"));
}
```

Using such a mechanism is cumbersome, however. To output a `Muffin` in the middle of another line of text, you'd need to split the line into two calls with a call to `Muffin::output()` in between, as shown in the following:

```
printf("The muffin is ");
myMuffin.output();
printf(" -- yummy!\n");
```

Overloading the `<<` operator lets you output a `Muffin` just like you output a `string` — by providing it as an argument to `<<`. Chapter 14 covers the details of overloading the `<<` and `>>` operators.

STRING STREAMS

String streams provide a way to use stream semantics with strings. In this way, you can have an *in-memory stream* that represents textual data. For example, in a GUI application you might want to use streams to build up textual data, but instead of outputting the text to the console or a file, you might want to display the result in a GUI element like a message box or an edit control. Another example could be that you want to pass a string stream around to different functions, while retaining the current read position, so that each function can process the next part of the stream. String streams are also useful for parsing text, because streams have built-in tokenizing functionality.

The `ostringstream` class is used to write data to a `string`, while the `istringstream` class is used to read data from a `string`. They are both defined in the `<sstream>` header file. Because `ostringstream` and `istringstream` inherit the same behavior as `ostream` and `istream`, working with them is pleasantly similar.

The following program requests words from the user and outputs them to a single `ostringstream`, separated by the tab character. At the end of the program, the whole stream is turned into a `string` object using the `str()` method and is written to the console. Input of tokens can be stopped by entering the token “done” or by closing the input stream with Control+D (Unix) or Control+Z (Windows):

```
cout << "Enter tokens. Control+D (Unix) or Control+Z (Windows) to end" << endl;
ostringstream outStream;
while (cin) {
    string nextToken;
    cout << "Next token: ";
    cin >> nextToken;
    if (nextToken == "done")
        break;
    outStream << nextToken << "\t";
}
cout << "The end result is: " << outStream.str();
```

Reading data from a string stream is similarly familiar. The following function creates and populates a `Muffin` object (see earlier example) from a string input stream. The stream data is in a fixed format so that the function can easily turn its values into calls to the `Muffin` setters:

```
Muffin createMuffin(istringstream& inStream)
{
    Muffin muffin;
    // Assume data is properly formatted:
    // Description size chips
    string description;
    int size;
    bool hasChips;
    // Read all three values. Note that chips is represented
    // by the strings "true" and "false"
    inStream >> description >> size >> boolalpha >> hasChips;
    muffin.setSize(size);
    muffin.setDescription(description);
    muffin.setHasChocolateChips(hasChips);
    return muffin;
}
```

NOTE *Turning an object into a “flattened” type, like a string, is often called marshalling. Marshalling is useful for saving objects to disk or sending them across a network.*

The main advantage of a string stream over a standard C++ string is that, in addition to data, the object knows where the next read or write operation will take place, also called current position. There may also be performance benefits depending on the particular implementation of string streams. For example, if you need to append a lot of strings together, it might be more efficient to use a string stream, instead of repeatedly calling the `+=` operator on a `string` object.

FILE STREAMS

Files lend themselves very well to the stream abstraction because reading and writing files always involves a position in addition to the data. In C++, the `ofstream` and `ifstream` classes provide output and input functionality for files. They are defined in the `<fstream>` header file.

When dealing with the file system, it is especially important to detect and handle error cases. The file you are working with could be on a network file store that just went offline, or you may be trying to write to a file that is located on a disk that is full. Maybe you are trying to open a file to which the current user does not have permissions to. Error conditions can be detected by using the standard error handling mechanisms described earlier.

The only major difference between output file streams and other output streams is that the file stream constructor can take the name of the file and the mode in which you would like to open it. The default mode is `write`, `ios_base::out`, which starts writing to a file at the beginning, overwriting any existing data. You can also open an output file stream in append mode by specifying the constant `ios_base::app` as second argument to the file stream constructor. The following table lists the different constants that are available:

CONSTANT	DESCRIPTION
<code>ios_base::app</code>	Open and go to the end before each write operation.
<code>ios_base::ate</code>	Open and go to the end immediately after opening.
<code>ios_base::binary</code>	Perform input and output in binary mode (as opposed to text mode).
<code>ios_base::in</code>	Open for input, start reading at the beginning.
<code>ios_base::out</code>	Open for output, start writing at the beginning, overwriting existing data.
<code>ios_base::trunc</code>	Open for output, delete all existing data (truncate).

The following program opens the file `test.txt` and outputs the arguments to the program. The `ifstream` and `ofstream` destructors automatically close the underlying file, so there is no need to explicitly call `close()`:

```
int main(int argc, char* argv[])
{
    ofstream outFile("test.txt", ios_base::trunc);
    if (!outFile.good())
        cerr << "Error while opening output file!" << endl;
```

```

        return -1;
    }
    outFile << "There were " << argc << " arguments to this program." << endl;
    outFile << "They are: " << endl;
    for (int i = 0; i < argc; i++) {
        outFile << argv[i] << endl;
    }
    return 0;
}

```

Jumping around with `seek()` and `tell()`

The `seek()` and `tell()` methods are present on all input and output streams, but they rarely make sense outside of the context of file streams.

The `seek()` methods let you move to an arbitrary position within an input or output stream. There are several forms of `seek()`. The methods of `seek()` within an input stream are actually called `seekg()` (the `g` is for `get`), and the versions of `seek()` in an output stream are called `seekp()` (the `p` is for `put`). You might wonder why there is both a `seekg()` and a `seekp()` method, instead of one `seek()` method. The reason is that you can have streams that are both input and output, for example, file streams. In that case, the stream needs to remember both a read position and a separate write position. This is also called bidirectional I/O and is covered later in this chapter.

There are two overloads of `seekg()` and two of `seekp()`. One overload accepts a single argument, an absolute position, and seeks to this absolute position. The second overload accepts an offset and a position, and seeks an offset relative to the given position. Positions are of type `ios_base::streampos`, while offsets are of type `ios_base::streamoff`, both are measured in bytes. There are three predefined positions available:

POSITION	DESCRIPTION
<code>ios_base::beg</code>	The beginning of the stream
<code>ios_base::end</code>	The end of the stream
<code>ios_base::cur</code>	The current position in the stream

For example, to seek to an absolute position in an output stream, you can use the one-parameter version of `seekp()`, as in the following case, which uses the constant `ios_base::beg` to move to the beginning of the stream:

```
outStream.seekp(ios_base::beg);
```

Seeking within an input stream is exactly the same, except that the `seekg()` method is used:

```
inStream.seekg(ios_base::beg);
```

The two-argument versions move to a relative position in the stream. The first argument prescribes how many positions to move and the second argument provides the starting point. To move relative to the beginning of the file, the constant `ios_base::beg` is used. To move relative to the end of the

file, `ios_base::end` is used. To move relative to the current position, `ios_base::cur` is used. For example, the following line moves to the second byte from the beginning of the stream. Note that integers are implicitly converted to type `ios_base::streampos` and `ios_base::streamoff`:

```
outStream.seekp(2, ios_base::beg);
```

The next example moves to the third-to-last byte of an input stream.

```
inStream.seekg(-3, ios_base::end);
```

You can also query a stream's current location using the `tell()` method which returns an `ios_base::streampos` that indicates the current position. You can use this result to remember the current marker position before doing a `seek()` or to query whether you are in a particular location. As with `seek()`, there are separate versions of `tell()` for input streams and output streams. Input streams use `tellg()`, and output streams use `tellp()`.

The following code checks the position of an input stream to determine if it is at the beginning.

```
ios_base::streampos curPos = inStream.tellg();
if (ios_base::beg == curPos) {
    cout << "We're at the beginning." << endl;
}
```

Following is a sample program that brings it all together. This program writes into a file called `test.out` and performs the following tests:

1. Outputs the string `12345` to the file.
2. Verifies that the marker is at position 5 in the stream.
3. Moves to position 2 in the output stream.
4. Outputs a `0` in position 2 and closes the output stream.
5. Opens an input stream on the `test.out` file.
6. Reads the first token as an integer.
7. Confirms that the value is `12045`.

```
ofstream fout("test.out");
if (!fout) {
    cerr << "Error opening test.out for writing" << endl;
    return 1;
}
// 1. Output the string "12345".
fout << "12345";
// 2. Verify that the marker is at position 5.
ios_base::streampos curPos = fout.tellp();
if (5 == curPos) {
    cout << "Test passed: Currently at position 5" << endl;
} else {
    cout << "Test failed: Not at position 5" << endl;
}
```

```

// 3. Move to position 2 in the stream.
fout.seekp(2, ios_base::beg);
// 4. Output a 0 in position 2 and close the stream.
fout << 0;
fout.close();
// 5. Open an input stream on test.out.
ifstream fin("test.out");
if (!fin) {
    cerr << "Error opening test.out for reading" << endl;
    return 1;
}
// 6. Read the first token as an integer.
int testVal;
fin >> testVal;
// 7. Confirm that the value is 12045.
const int expected = 12045;
if (testVal == expected) {
    cout << "Test passed: Value is " << expected << endl;
} else {
    cout << "Test failed: Value is not " << expected
        << " (it was " << testVal << ")" << endl;
}

```

Linking Streams Together

A link can be established between any input and output streams to give them *flush-on-access* behavior. In other words, when data is requested from an input stream, its linked output stream will automatically flush. This behavior is available to all streams, but is particularly useful for file streams that may be dependent upon each other.

Stream linking is accomplished with the `tie()` method. To tie an output stream to an input stream, call `tie()` on the input stream, and pass the address of the output stream. To break the link, pass `nullptr`.

The following program ties the input stream of one file to the output stream of an entirely different file. You could also tie it to an output stream on the same file, but bidirectional I/O (covered below) is perhaps a more elegant way to read and write the same file simultaneously.

```

ifstream inFile("input.txt"); // Note: input.txt must exist.
ofstream outFile("output.txt");
// Set up a link between inFile and outFile.
inFile.tie(&outFile);
// Output some text to outFile. Normally, this would
// not flush because std::endl is not sent.
outFile << "Hello there!";
// outFile has NOT been flushed.
// Read some text from inFile. This will trigger flush()
// on outFile.
string nextToken;
inFile >> nextToken;
// outFile HAS been flushed.

```

The `flush()` method is defined on the `ostream` base class, so you can also link an output stream to another output stream:

```
outFile.tie(&anotherOutputFile);
```

Such a relationship would mean that every time you wrote to one file, the buffered data that had been sent to the other file would be written. You could use this mechanism to keep two related files synchronized.

One example of this stream linking is the link between `cout` and `cin`. Whenever you try to input data from `cin`, `cout` is automatically flushed.

BIDIRECTIONAL I/O

So far, this chapter has discussed input and output streams as two separate but related classes. In fact, there is such a thing as a stream that performs both input and output. A *bidirectional stream* operates as both an input stream and an output stream.

Bidirectional streams are deriving from `iostream`, which in turn derives from both `istream` and `ostream`, thus serving as an example of useful multiple inheritance. As you would expect, bidirectional streams support both the `>>` operator and the `<<` operator, as well as the methods of both input streams and output streams.

The `fstream` class provides a bidirectional file stream. `fstream` is ideal for applications that need to replace data within a file because you can read until you find the correct position, then immediately switch to writing. For example, imagine a program that stores a list of mappings between ID numbers and phone numbers. It might use a data file with the following format:

```
123 408-555-0394
124 415-555-3422
263 585-555-3490
100 650-555-3434
```

A reasonable approach to such a program would be to read in the entire data file when the program opens and rewrite the file, with any modifications, when the program closes. If the data set is huge, however, you might not be able to keep everything in memory. With `iostreams`, you don't have to. You can easily scan through the file to find a record, and you can add new records by opening the file for output in append mode. To modify an existing record, you could use a bidirectional stream, as in the following function that changes the phone number for a given ID:

```
bool changeNumberForID(const string& inFile, int inID,
    const string& inNewNumber)
{
    fstream ioData(inFile.c_str());
    if (!ioData) {
        cerr << "Error while opening file " << inFile << endl;
        return false;
    }
```

```
// Loop until the end of file
while (ioData.good()) {
    int id;
    string number;
    // Read the next ID.
    ioData >> id;
    // Check to see if the current record is the one being changed.
    if (id == inID) {
        // Seek the write position to the current read position
        ioData.seekp(ioData.tellg());
        // Output a space, then the new number.
        ioData << " " << inNewNumber;
        break;
    }
    // Read the current number to advance the stream.
    ioData >> number;
}
return true;
}
```

Of course, an approach like this will work properly only if the data is of a fixed size. When the preceding program switched from reading to writing, the output data overwrote other data in the file. To preserve the format of the file, and to avoid writing over the next record, the data had to be the exact same size.

String streams can also be accessed in a bidirectional manner through the `stringstream` class.

NOTE *Bidirectional streams have separate pointers for the read position and the write position. When switching between reading and writing, you will need to seek to the appropriate position.*

SUMMARY

Streams provide a flexible and object-oriented way to perform input and output. The most important message in this chapter, even more important than the use of streams, is the concept of a stream. Some operating systems may have their own file access and I/O facilities, but knowledge of how streams and stream-like libraries work is essential to working with any type of modern I/O system.

13

Handling Errors

WHAT'S IN THIS CHAPTER?

- How to handle errors in C++, including pros and cons of exceptions
- The syntax of exceptions
- Exception class hierarchies and polymorphism
- Stack unwinding and cleanup
- Common error-handling situations

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Inevitably, your C++ programs will encounter errors. The program might be unable to open a file, the network connection might go down, or the user might enter an incorrect value, to name a few possibilities. The C++ language provides a feature called *exceptions* to handle these *exceptional* but not *unexpected* situations.

The code examples in this book so far have virtually always ignored error conditions for brevity. This chapter rectifies that simplification by teaching you how to incorporate error handling into your programs from their beginnings. It focuses on C++ exceptions, including the details of their syntax, and describes how to employ them effectively to create well-designed error-handling programs.

ERRORS AND EXCEPTIONS

No program exists in isolation; they all depend on external facilities such as interfaces with the operating system, networks and file systems, external code such as third-party libraries, and user input. Each of these areas can introduce situations which require responding to problems which may be encountered. These potential problems can be referred to with the general term *exceptional situations*. Even perfectly written programs encounter errors and exceptional situations. Thus, anyone who writes a computer program must include error-handling capabilities. Some languages, such as C, do not include many specific language facilities for error handling. Programmers using these languages generally rely on return values from functions and other *ad hoc* approaches. Other languages, such as Java, enforce the use of a language feature called *exceptions* as an error-handling mechanism. C++ lies between these extremes. It provides language support for exceptions, but does not require their use. However, you can't ignore exceptions entirely in C++ because a few basic facilities, such as memory allocation routines, use them.

What Are Exceptions, Anyway?

Exceptions are a mechanism for a piece of code to notify another piece of code of an “exceptional” situation or error condition without progressing through the normal code paths. The code that encounters the error *throws* the exception, and the code that handles the exception *catches* it. Exceptions do not follow the fundamental rule of step-by-step execution to which you are accustomed. When a piece of code throws an exception, the program control immediately stops executing code step by step and transitions to the exception handler, which could be anywhere from the next line in the same function to several function calls up the stack. If you like sports analogies, you can think of the code that throws an exception as an outfielder throwing a baseball back to the infield, where the nearest infielder (closest exception handler) catches it. Figure 13-1 shows a hypothetical stack of three function calls. Function A() has the exception handler. It calls function B(), which calls function C(), which throws the exception.

Figure 13-2 shows the handler catching the exception. The stack frames for C() and B() have been removed, leaving only A().

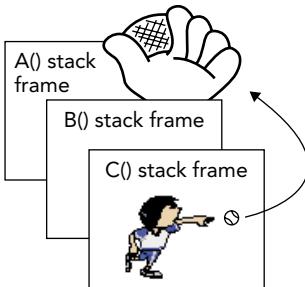


FIGURE 13-1

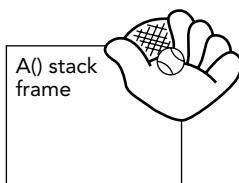


FIGURE 13-2

Most modern programming languages, such as C# and Java, have support for exceptions, so it's no surprise that C++ has full-fledged support for them as well. However, if you are coming from C then exceptions are something new, but once you get used to them you probably don't want to go back.

Why Exceptions in C++ Are a Good Thing

As mentioned earlier, run-time errors in programs are inevitable. Despite that fact, error handling in most C and C++ programs is messy and *ad hoc*. The *de facto* C error-handling standard, which was carried over into many C++ programs, uses integer function return codes and the `errno` macro to signify errors. Each thread has its own `errno` value. `errno` acts as a thread-local integer variable that functions can use to communicate errors back to calling functions.

Unfortunately, the integer return codes and `errno` are used inconsistently. Some functions might choose to return 0 for success and -1 for an error. If they return -1, they also set `errno` to an error code. Other functions return 0 for success and nonzero for an error, with the actual return value specifying the error code. These functions do not use `errno`. Still others return 0 for failure instead of for success, presumably because 0 always evaluates to `false` in C and C++.

These inconsistencies can cause problems because programmers encountering a new function often assume that its return codes are the same as other similar functions. That is not always true.

On Solaris 9, there are two different libraries of synchronization objects: the POSIX version and the Solaris version. The function to initialize a semaphore in the POSIX version is called `sem_init()`, and the function to initialize a semaphore in the Solaris version is called `sema_init()`. As if that weren't confusing enough, the two functions handle error codes differently! `sem_init()` returns -1 and sets `errno` on error, while `sema_init()` returns the error code directly as a positive integer, and does not set `errno`.

Another problem is that the return type of functions in C++ can only be of one type, so if you need to return both an error and a value, you must find an alternative mechanism. One solution is to return a `std::pair` or `std::tuple`, an object that you can use to store two or more types. They are discussed in later STL chapters. Another choice is to define your own struct or class that contains several values, and return an instance of that struct or class from your function. Yet another option is to return the value or error through a reference parameter or to make the error code one possible value of the return type, such as a `nullptr` pointer. In all these solutions, the caller is responsible to explicitly check for any errors returned from the function and if it doesn't handle the error itself, it should propagate the error to its caller. Unfortunately, this will often result in the loss of critical details about the error.

C programmers may be familiar with a mechanism known as `setjmp()`/`longjmp()`. This mechanism cannot be used correctly in C++, because it bypasses scoped destructors on the stack. You should avoid it at all cost, even in C programs; therefore this book does not explain the details of how to use it.

Exceptions provide an easier, more consistent, and safer mechanism for error handling. There are several specific advantages of exceptions over the *ad hoc* approaches in C and C++.

- When return codes are used as an error reporting mechanism, you might forget to check the return code and properly handle it either locally or by propagating it upwards. Exceptions cannot be forgotten or ignored: If your program fails to catch an exception, it will terminate.
- When integer return codes are used, they generally do not contain sufficient information. You can use exceptions to pass as much information as you want from the code that

finds the error to the code that handles it. Exceptions can also be used to communicate information other than errors, though many programmers consider that an abuse of the exception mechanism.

- Exception handling can skip levels of the call stack. That is, a function can handle an error that occurred several function calls down the stack, without error-handling code in the intermediate functions. Return codes require each level of the call stack to clean up explicitly after the previous level.

In some compilers (fewer and fewer these days), exception handling added a tiny amount of overhead to any function that had an exception handler. For most modern compilers there is a trade-off in that there is almost no overhead in the non-throwing case, and only some overhead when you actually throw something. This trade-off is not a bad thing because throwing an exception should be exceptional. Do not assume that the urban legend about exceptions introducing serious overhead is true; you have to check it out in your compiler.

Exception handling is not enforced in C++. For example, in Java a function that does not specify a list of possible exceptions that it can throw is not allowed to throw any exceptions. In C++, it is just the opposite: a function that does not specify a list of exceptions can throw any exception it wants! Additionally, the exception list is not enforced at compile time in C++, meaning that the exception list of a function can be violated at run time. Some tools, such as using /Analyze with Microsoft VC++, will check exceptions and report potential problems. Note that since C++11, throw lists are deprecated.

Recommendation

I recommend exceptions as a useful mechanism for error handling. I feel that the structure and error-handling formalization that exceptions provide outweigh the less desirable aspects. Thus, the remainder of this chapter focuses on exceptions. Also, many popular libraries, such as the STL and Boost use exceptions, so you need to be prepared to handle them.

EXCEPTION MECHANICS

Exceptional situations arise frequently in file input and output. The following is a function to open a file, read a list of integers from the file, and store the integers in the supplied `std::vector` data structure. The lack of error handling should jump out at you.

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());

    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
}
```

The following line keeps reading values from the `ifstream` until the end of the file is reached or until an error occurs.

```
while (istr >> temp) {
```

This works because the `>>` operator returns a reference to the `ifstream` object itself. Additionally, `ifstream` provides a `bool()` conversion operator implemented as follows:

```
return !fail();
```

If the `>>` operator encounters an error, it will set the `fail` bit of the `ifstream` object. In that case, the `bool()` conversion operator will return `false` and the `while` loop will terminate. Streams are discussed in more detail in Chapter 12.

You might use `readIntegerFile()` like this:

```
vector<int> myInts;
const string fileName = "IntegerFile.txt";
readIntegerFile(fileName, myInts);
for (const auto element : myInts) {
    cout << element << " ";
}
cout << endl;
```

The rest of this section shows you how to add error handling with exceptions.

Throwing and Catching Exceptions

Using exceptions consists of providing two parts in your program: a `try/catch` construct, to handle an exception, and a `throw` statement, that throws an exception. Both must be present in some form to make exceptions work. However, in many cases, the `throw` happens deep inside some library (including the C++ runtime) and the programmer never sees it, but still has to react to it using a `try/catch` construct.

The `try/catch` construct looks as follows:

```
try {
    // ... code which may result in an exception being thrown
} catch (exception-type1 exception-name) {
    // ... code which responds to the exception of type 1
} catch (exception-type2 exception-name) {
    // ... code which responds to the exception of type 2
}
// ... remaining code
```

The code which may result in an exception being thrown might contain a `throw` directly, or might be calling a function which either directly throws an exception or calls, by some unknown number of layers of calls, a function which throws an exception.

If no exception is thrown, the code in the `catch` blocks is not executed, and the “remaining code” which follows will follow the last statement executed in the `try` block.

If an exception is thrown, any code following the throw or following the call which resulted in the throw, is not executed, but control immediately goes to the right `catch` block depending on the type of the exception that is thrown.

If the `catch` block does not do a control transfer, for example by returning a value, throwing a new exception or rethrowing the exception, then the “remaining code” is executed after the last statement of that `catch` block.

The simplest example to demonstrate exception handling is avoiding divide-by-zero. This example throws an exception of type `std::invalid_argument` which requires the `<stdexcept>` header.

```
int SafeDivide(int num, int den)
{
    if (den == 0)
        throw invalid_argument("Divide by zero");
    return num / den;
}
int main()
{
    try {
        cout << SafeDivide(5, 2) << endl;
        cout << SafeDivide(10, 0) << endl;
        cout << SafeDivide(3, 3) << endl;
    } catch (const invalid_argument& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

The output is as follows:

```
2
Caught exception: Divide by zero
```

`throw` is a keyword in C++, and is the only way to throw an exception. The `invalid_argument()` part of the `throw` line means that you are constructing a new object of type `invalid_argument` to throw. It is one of the standard exceptions provided by the C++ Standard Library. All Standard Library exceptions form a hierarchy, which is discussed later in this chapter. Each class in the hierarchy supports a `what()` method that returns a `const char*` string describing the exception. This is the string you provide in the constructor of the exception.

The `throw` keyword can also be used to rethrow the current exception. For example:

```
void g() { throw invalid_argument("Some exception"); }
void f()
{
    try {
        g();
    } catch (const invalid_argument& e) {
        cout << "caught in f: " << e.what() << endl;
        throw; // rethrow
    }
}
```

```

int main()
{
    try {
        f();
    } catch (const invalid_argument& e) {
        cout << "caught in main: " << e.what() << endl;
    }
    return 0;
}

```

This example produces the following output:

```

caught in f: Some exception
caught in main: Some exception

```

Let's go back to the `readIntegerFile()` function. The most likely problem to occur is for the file open to fail. That's a perfect situation for throwing an exception. This code throws an exception of type `std::exception` which requires the `<exception>` header. The syntax looks like this:

```

#include <exception>
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw exception();
    }
    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
}

```

If the function fails to open the file and executes the `throw exception();` line, the rest of the function is skipped, and control transitions to the nearest exception handler.

Throwing exceptions in your code is most useful when you also write code that handles them. Exception handling is a way to “try” a block of code, with another block of code designated to react to any problems that might occur. In the following `main()` function the `catch` statement reacts to any exception of type `exception` that was thrown within the `try` block by printing an error message. If the `try` block finishes without throwing an exception, the `catch` block is skipped. You can think of `try/catch` blocks as glorified `if` statements. If an exception is thrown in the `try` block, execute the `catch` block. Otherwise, skip it.

```

int main()
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";
    try {
        readIntegerFile(fileName, myInts);
    } catch (const exception& e) {

```

```

        cerr << "Unable to open file " << fileName << endl;
        return 1;
    }
    for (const auto element : myInts) {
        cout << element << " ";
    }
    cout << endl;
    return 0;
}

```

NOTE *Although by default, streams do not throw exceptions, you can tell the streams to throw exceptions for error conditions by calling their exceptions() method. However, not all compilers give useful information in the stream exceptions they throw. For those compilers it might be better to deal with the stream state directly instead of using exceptions. This book does not use stream exceptions.*

Exception Types

You can throw an exception of any type. The preceding example throws an object of type exception, but exceptions do not need to be objects. You could throw a simple int like this:

```

void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw 5;
    }

    // Read the integers one-by-one and add them to the vector
    while (istr >> temp) {
        dest.push_back(temp);
    }
}

```

You would then need to change the catch statement as well:

```

try {
    readIntegerFile(fileName, myInts);
} catch (int e) {
    cerr << "Unable to open file " << fileName << " (" << e << ")" << endl;
    return 1;
}

```

Alternatively, you could throw a `const char*` C-style string. This technique is sometimes useful because the string can contain information about the exception.

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw "Unable to open file";
    }

    // Read the integers one-by-one and add them to the vector
    while (istr >> temp) {
        dest.push_back(temp);
    }
}
```

When you catch the `const char*` exception, you can print the result:

```
try {
    readIntegerFile(fileName, myInts);
} catch (const char* e) {
    cerr << e << endl;
    return 1;
}
```

Despite the previous examples, you should generally throw objects as exceptions for two reasons:

- Objects convey information by their class name.
- Objects can store information, including strings that describe the exceptions.

The C++ standard library defines a number of predefined exception classes and you can write your own exception classes. Details are described later in this chapter.

Catching Exception Objects by `const` and Reference

In the preceding example in which `readIntegerFile()` throws an object of type `exception`, the `catch` line looks like this:

```
} catch (const exception& e) {
```

However, there is no requirement to catch objects by `const` reference. You could catch the object by value like this:

```
} catch (exception e) {
```

Alternatively, you could catch the object by reference (without the `const`):

```
} catch (exception& e) {
```

Also, as you saw in the `const char*` example, you can catch pointers to exceptions, as long as pointers to exceptions are thrown.

NOTE *It is recommended to catch exceptions by const reference. This avoids object slicing which could happen when you catch exceptions by value.*

Throwing and Catching Multiple Exceptions

Failure to open the file is not the only problem `readIntegerFile()` could encounter. Reading the data from the file can cause an error if it is formatted incorrectly. Here is an implementation of `readIntegerFile()` that throws an exception if it cannot either open the file or read the data correctly:

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw runtime_error("Unable to open the file.");
    }
    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
    if (!istr.eof()) {
        // We did not reach the end-of-file.
        // This means that some error occurred while reading the file.
        // Throw an exception.
        throw runtime_error("Error reading the file.");
    }
}
```

Your code in `main()` does not need to change because it already catches an exception of type `exception`, from which `runtime_error` derives. However, that exception could now be thrown in two different situations.

```
try {
    readIntegerFile(fileName, myInts);
} catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
}
```

Alternatively, you could throw two different types of exceptions from `readIntegerFile()`. Here is an implementation of `readIntegerFile()` that throws an exception object of class `invalid_argument` if the file cannot be opened and an object of class `runtime_error` if the integers cannot be read. Both `invalid_argument` and `runtime_error` are classes defined in the header file `<stdexcept>` as part of the C++ Standard Library.

```

#include <stdexcept>
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw invalid_argument("Unable to open the file.");
    }
    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
    if (!istr.eof()) {
        // We did not reach the end-of-file.
        // This means that some error occurred while reading the file.
        // Throw an exception.
        throw runtime_error("Error reading the file.");
    }
}

```

There are no public default constructors for `invalid_argument` and `runtime_error`, only string constructors.

Now `main()` can catch both `invalid_argument` and `runtime_error` with two `catch` statements:

```

try {
    readIntegerFile(fileName, myInts);
} catch (const invalid_argument& e) {
    cerr << e.what() << endl;
    return 1;
} catch (const runtime_error& e) {
    cerr << e.what() << endl;
    return 1;
}

```

If an exception is thrown inside the `try` block, the compiler will match the type of the exception to the proper `catch` handler. So, if `readIntegerFile()` is unable to open the file and throws an `invalid_argument` object, it will be caught by the first `catch` statement. If `readIntegerFile()` is unable to read the file properly and throws a `runtime_error`, then the second `catch` statement will catch the exception.

Matching and `const`

The `const`-ness specified in the type of the exception you want to catch makes no difference for matching purposes. That is, this line matches any exception of type `runtime_error`.

```
    } catch (const runtime_error& e) {
```

The following line also matches any exception of type `runtime_error`:

```
    } catch (runtime_error& e) {
```

Matching Any Exception

You can write a `catch` line that matches any possible exception with the special syntax shown in the following example:

```
try {
    readIntegerFile(fileName, myInts);
} catch (...) {
    cerr << "Error reading or opening file " << fileName << endl;
    return 1;
}
```

The three dots are not a typo. They are a wildcard that match any exception type. When you are calling poorly documented code, this technique can be useful to ensure that you catch all possible exceptions. However, in situations where you have complete information about the set of thrown exceptions, this technique is not recommended because it handles every exception type identically. It's better to match exception types explicitly and take appropriate, targeted action.

A possible use of a `catch` block matching any exception is as a default `catch` handler. When an exception is thrown, a `catch` handler is looked up in the order that they appear in the code. The following example shows how you can write `catch` handlers that explicitly handle `invalid_argument` and `runtime_error` exceptions and how to include a default `catch` handler for all other exceptions.

```
try {
    // Code that can throw exceptions
} catch (const invalid_argument& e) {
    // Handle invalid_argument exception
} catch (const runtime_error& e) {
    // Handle runtime_error exception
} catch (...) {
    // Handle all other exceptions
}
```

Uncaught Exceptions

If your program throws an exception that is not caught anywhere, the program will terminate. Basically there is a `try/catch` construct around the call to your `main()` function which catches all unhandled exceptions, something as follows:

```
try {
    main(argc, argv);
} catch (...) {
    // issue error message and terminate program
}
```

However, this behavior is not usually what you want. The point of exceptions is to give your program a chance to handle and correct undesirable or unexpected situations.

WARNING *Catch and handle all possible exceptions thrown in your programs.*

Even if you can't handle a particular exception, you should still write code to catch it and print or show an appropriate error message.

It is also possible to change the behavior of your program if there is an uncaught exception. When the program encounters an uncaught exception, it calls the built-in `terminate()` function, which calls `abort()` from `<cstdlib>` to kill the program. You can set your own `terminate_handler` by calling `set_terminate()` with a pointer to a callback function that takes no arguments and returns no value. `terminate()`, `set_terminate()`, and `terminate_handler` are all declared in the `<exception>` header. The following code shows a high-level overview of how it works.

```
try {
    main(argc, argv);
} catch (...) {
    if (terminate_handler != nullptr) {
        terminate_handler();
    } else {
        terminate();
    }
}
// normal termination code
```

Before you get too excited about this feature, you should know that your callback function must still terminate the program. It can't just ignore the error. However, you can use it to print a helpful error message before exiting. Here is an example of a `main()` function that doesn't catch the exceptions thrown by `readIntegerFile()`. Instead, it sets the `terminate_handler` to a callback that prints an error message before exiting:

```
void myTerminate()
{
    cout << "Uncaught exception!" << endl;
    exit(1);
}
int main()
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";
    set_terminate(myTerminate);
    readIntegerFile(fileName, myInts);
    for (const auto element : myInts) {
        cout << element << " ";
    }
    cout << endl;
    return 0;
}
```

Although not shown in this example, `set_terminate()` returns the old `terminate_handler` when it sets the new one. The `terminate_handler` applies program-wide, so it's considered good style to reset the old `terminate_handler` when you have completed the code that needed the new `terminate_handler`. In this case, the entire program needs the new `terminate_handler`, so there's no point in resetting it.

Although it's important to know about `set_terminate()`, it's not a very effective exception-handling approach. It's recommended to try to catch and handle each exception individually in order to provide more precise error handling.

Throw Lists

Older versions of C++ allowed you to specify the exceptions a function or method intends to throw. This specification is called the *throw list* or the *exception specification*. Since C++11, these throw lists are deprecated. However, the syntax is still explained here because you might encounter it in existing code bases. Here is the `readIntegerFile()` function from the earlier example with the proper throw list:

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw(invalid_argument, runtime_error)
{
    // Remainder of the function is the same as before
}
```

The throw list shows the types of exceptions that can be thrown from the function. Note that the throw list must also be provided for the function prototype:

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw(invalid_argument, runtime_error);
```

You cannot overload a function based solely on different exceptions in the throw list.

If a function or method specifies no throw list, it can throw any exception. You've already seen this behavior in the previous implementation of the `readIntegerFile()` function. If you want to specify that a function or method throws no exceptions, you need to use `noexcept` (since C++11) or `throw()` (deprecated) as follows:

```
void readIntegerFile(const string& fileName, vector<int>& dest) noexcept,
// or
void readIntegerFile(const string& fileName, vector<int>& dest) throw();
```

NOTE A function without a throw list can throw exceptions of any type. A function with `noexcept` shouldn't throw any exception.

If this behavior seems backward to you, you're not alone. However, it's best just to accept it and move on.

WARNING *Since C++11, any exception specification is deprecated, except for noexcept.*

Unexpected Exceptions

Unfortunately, the throw list is not enforced at compile time in C++. Code that calls `readIntegerFile()` does not need to catch the exceptions listed in the throw list. This behavior is different from that in other languages, such as Java, which requires a function or method to catch exceptions or declare them in their own function or method throw lists.

Additionally, you could implement `readIntegerFile()` like this:

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw(invalid_argument, runtime_error)
{
    throw 5;
}
```

Even though the throw list states that `readIntegerFile()` doesn't throw an `int`, this code, which obviously throws an `int`, compiles and runs. However, it won't do what you want. Suppose that you write this `main()` function which has a `catch` block for `int`:

```
int main()
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";
    try {
        readIntegerFile(fileName, myInts);
    } catch (int x) {
        cerr << "Caught int " << x << endl;
    }
    return 0;
}
```

When this program runs and `readIntegerFile()` throws the `int` exception, the program terminates. It does not allow `main()` to catch the `int`.

WARNING *Throw lists don't prevent functions from throwing unlisted exception types, but they prevent the exception from leaving the function, resulting in a run-time error.*

NOTE All versions of Microsoft Visual C++, up to version 2013 at the time of this writing, do not support throw lists for functions as explained earlier. As a result, the preceding `main()` function will catch the `int` exception because Visual C++ just ignores the `throw(invalid_argument, runtime_error)` specification and will issue a warning like “warning C4290: C++ exception specification ignored except to indicate a function is not `__declspec(nothrow)`”. This is fine because C++11 has deprecated throw lists.

When a function marked as `noexcept` throws an exception, C++ calls `terminate()` to terminate the application. When a function throws an exception that is not listed in its throw list, C++ calls a special function `unexpected()`. The built-in implementation of `unexpected()` calls `terminate()`. However, just as you can set your own `terminate_handler`, you can set your own `unexpected_handler`. Unlike in the `terminate_handler`, you can actually do something other than just terminate the program in the `unexpected_handler`. Your version of the function must either throw a new exception or terminate the program — it can't just exit the function normally. If it throws a new exception, that exception will be substituted for the unexpected exception as if the new one had been thrown originally. If this substituted exception is also not listed in the throw list, the program will do one of two things. If the throw list for the function specifies `bad_exception`, then `bad_exception` will be thrown. Otherwise, the program will terminate. Custom implementations of `unexpected()` are normally used to convert unexpected exceptions into expected exceptions. For example, you could write a version of `unexpected()` like this:

```
void myUnexpected()
{
    cerr << "Unexpected exception!" << endl;
    throw runtime_error("");
}
```

This code converts an unexpected exception to a `runtime_error` exception, which the function `readIntegerFile()` has in its throw list.

You could set this `unexpected` exception handler in `main()` with the `set_unexpected()` function. Like `set_terminate()`, `set_unexpected()` returns the current handler. The `unexpected()` function applies program-wide, not just to this function, so you should reset the handler when you are done with the code that needed your special handler:

```
int main()
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";
    unexpected_handler old_handler = set_unexpected(myUnexpected);
    try {
        readIntegerFile(fileName, myInts);
    } catch (const invalid_argument& e) {
        cerr << e.what() << endl;
        return 1;
    }
}
```

```

} catch (const runtime_error& e) {
    cerr << e.what() << endl;
    return 1;
} catch (int x) {
    cerr << "Caught int " << x << endl;
}
set_unexpected(old_handler);
// Remainder of function omitted
}

```

Now `main()` handles any exception thrown from `readIntegerFile()` by converting it to a `runtime_error`. However, as with `set_terminate()`, it's recommended to use this capability judiciously. `unexpected()`, `set_unexpected()`, and `bad_exception` are all declared in the `<exception>` header file.

Changing the Throw List in Overridden Methods

When you override a virtual method in a derived class, you can change the throw list as long as you make it *more restrictive* than the throw list in the base class. The following changes qualify as more restrictive:

- Removing exceptions from the list
- Adding derived classes of exceptions that appear in the base class throw list
- Making it a `noexcept` method

The following changes do not qualify as more restrictive:

- Adding exceptions to the list that are not derived classes of exceptions in the base class throw list
- Removing the throw list entirely

WARNING *If you change throw lists when you override methods, remember that any code that called the base class version of the method must be able to call the derived class version. Thus, you can't add exceptions.*

For example, suppose that you have the following base class:

```

class Base
{
public:
    virtual ~Base() {}
    virtual void func() throw(exception) { cout << "Base!\n"; }
};

```

You could write a derived class that overrides `func()` and specifies that it doesn't throw any exceptions:

```
class Derived : public Base
{
public:
    virtual void func() noexcept { cout << "Derived!\n"; }
};
```

You could also override `func()` such that it throws a `runtime_error` as well as an exception, because `runtime_error` is a derived class of `exception`.

```
class Derived : public Base
{
public:
    virtual void func() throw(exception, runtime_error)
        { cout << "Derived!\n"; }
};
```

However, you cannot remove the throw list entirely, because that means `func()` could throw any exception.

As a second example, suppose `Base` looked like this:

```
class Base
{
public:
    virtual ~Base() {}
    virtual void func() throw(runtime_error) { cout << "Base!\n"; }
};
```

Then you cannot override `func()` in `Derived` with a throw list like this:

```
class Derived : public Base
{
public:
    virtual void func() throw(exception) // ERROR!
        { cout << "Derived!\n"; }
};
```

`exception` is a base class of `runtime_error`, so you cannot substitute an `exception` for a `runtime_error`.

Are Throw Lists Useful?

The exceptions thrown from a particular function are an important part of its interface, and should be documented as well as possible. In the past you could consider documenting them with throw lists. However, most of the existing C++ code, including the Standard Library, did not use throw lists. That made it difficult for you to determine which exceptions could be thrown when you used that code. Additionally, it is impossible to specify the exception characteristics

of templated functions and methods. When you don't even know what types will be used to instantiate the template, you have no way to determine the exceptions that methods of those types can throw.

Since C++11, the decision is easy; Never use throw lists because they are now deprecated, except for noexcept. Instead, document the possible exceptions a function can throw in its code documentation.

EXCEPTIONS AND POLYMORPHISM

As described earlier, you can actually throw any type of exception. However, classes are the most useful types of exceptions. In fact, exception classes are usually written in a hierarchy, so that you can employ polymorphism when you catch the exceptions.

The Standard Exception Hierarchy

You've already seen several exceptions from the C++ standard exception hierarchy: exception, runtime_error, and invalid_argument. Figure 13-3 shows the complete hierarchy:

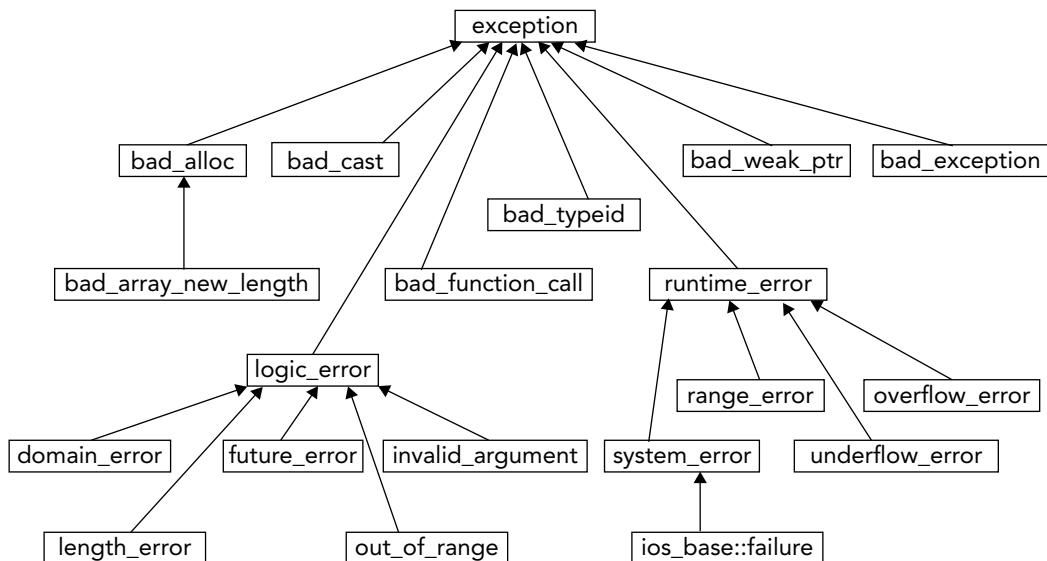


FIGURE 13-3

All of the exceptions thrown by the C++ Standard Library are objects of classes in this hierarchy. Each class in the hierarchy supports a `what()` method that returns a `const char*` string describing the exception. You can use this string in an error message.

All the exception classes except for the base `exception` require you to set in the constructor the string that will be returned by `what()`. That's why you have to specify a string in the constructors for `runtime_error` and `invalid_argument`. This has already been done in examples throughout this

chapter. Here is another version of `readIntegerFile()` that also includes the filename in the error message.

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        string error = "Unable to open file " + fileName;
        throw invalid_argument(error);
    }
    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
    if (!istr.eof()) {
        // We did not reach the end-of-file.
        // This means that some error occurred while reading the file.
        // Throw an exception.
        string error = "Unable to read file " + fileName;
        throw runtime_error(error);
    }
}
int main()
{
    // Code omitted
    try {
        readIntegerFile(fileName, myInts);
    } catch (const invalid_argument& e) {
        cerr << e.what() << endl;
        return 1;
    } catch (const runtime_error& e) {
        cerr << e.what() << endl;
        return 1;
    }
    // Code omitted
}
```

Catching Exceptions in a Class Hierarchy

A feature of exception hierarchies is that you can catch exceptions polymorphically. For example, if you look at the two `catch` statements in `main()` following the call to `readIntegerFile()`, you can see that they are identical except for the exception class that they handle. Conveniently, `invalid_argument` and `runtime_error` are both derived classes of `exception`, so you can replace the two `catch` statements with a single `catch` statement for class `exception`:

```
try {
    readIntegerFile(fileName, myInts);
} catch (const exception& e) {
```

```

    cerr << e.what() << endl;
    return 1;
}

```

The `catch` statement for an `exception` reference matches any derived classes of `exception`, including both `invalid_argument` and `runtime_error`. Note that the higher in the exception hierarchy that you catch exceptions, the less specific is your error handling. You should generally catch exceptions at as specific a level as possible.

WARNING *When you catch exceptions polymorphically, make sure to catch them by reference. If you catch exceptions by value, you can encounter slicing, in which case you lose information from the object. See Chapter 9 for details on slicing.*

When more than one `catch` clause is used, the `catch` clauses are matched in syntactic order as they appear in your code; the first one that matches, wins. If one `catch` is more inclusive than a later one, it will match first, and the more restrictive one, which comes later, will not be executed at all. Therefore, you should place your `catch` clauses from most restrictive to least restrictive order. For example, suppose that you want to catch `invalid_argument` from `readIntegerFile()` explicitly, but leave the generic `exception` match for any other exceptions. The correct way to do so is like this:

```

try {
    readIntegerFile(fileName, myInts);
} catch (const invalid_argument& e) { // List the derived class first.
    // Take some special action for invalid filenames.
} catch (const exception& e) { // Now list exception
    cerr << e.what() << endl;
    return 1;
}

```

The first `catch` statement catches `invalid_argument` exceptions, and the second catches any other exceptions of type `exception`. However, if you reverse the order of the `catch` statements, you don't get the same result:

```

try {
    readIntegerFile(fileName, myInts);
} catch (const exception& e) { // BUG: catching base class first!
    cerr << e.what() << endl;
    return 1;
} catch (const invalid_argument& e) {
    // Take some special action for invalid filenames.
}

```

With this order, any exception of a class that derives from `exception` is caught by the first `catch` statement; the second will never be reached. Some compilers issue a warning in this case, but you shouldn't count on it.

Writing Your Own Exception Classes

There are two advantages to writing your own exception classes.

1. The number of exceptions in the C++ Standard Library is limited. Instead of using an exception class with a generic name, such as `runtime_error`, you can create classes with names that are more meaningful for the particular errors in your program.
2. You can add your own information to these exceptions. The exceptions in the standard hierarchy allow you to set only an error string. You might want to pass different information in the exception.

It's recommended that all the exception classes that you write inherit directly or indirectly from the standard exception class. If everyone on your project follows that rule, you know that every exception in the program will be derived from `exception` (assuming that you aren't using third-party libraries that break this rule). This guideline makes exception handling via polymorphism significantly easier.

For example, `invalid_argument` and `runtime_error` don't capture very well the file opening and reading errors in `readIntegerFile()`. You can define your own error hierarchy for file errors, starting with a generic `FileError` class:

```
class FileError : public exception
{
public:
    FileError(const string& fileIn) : mFile(fileIn) {}
    virtual const char* what() const noexcept override { return mMsg.c_str(); }
    const string& getFileName() { return mFile; }
protected:
    void setMessage(const string& message) { mMsg = message; }
private:
    string mFile, mMsg;
};
```

As a good programming citizen, you should make `FileError` a part of the standard exception hierarchy. It seems appropriate to integrate it as a child of `exception`. When you derive from `exception`, you can override the `what()` method, which has the prototype shown and which must return a `const char*` string that is valid until the object is destroyed. In the case of `FileError`, this string comes from the `mMsg` data member, which is set to "" in the constructor. Derived classes of `FileError` must set this `mMsg` string to something different if they want a different message.

The generic `FileError` class also contains a filename, a public accessor for that filename, and a protected setter so that derived classes can set the message.

The first exceptional situation in `readIntegerFile()` occurs if the file cannot be opened. Thus, you might want to write a `FileOpenError` derived from `FileError`:

```
class FileOpenError : public FileError
{
public:
    FileOpenError(const string& fileNameIn);
```

```
FileOpenError::FileOpenError(const string& fileNameIn) : FileError(fileNameIn)
{
    setMessage("Unable to open " + fileNameIn);
}
```

The `FileOpenError` changes the `mMsg` string to represent the file-opening error.

The second exceptional situation in `readIntegerFile()` occurs if the file cannot be read properly. It might be useful for this exception to contain the line number of the error in the file, as well as the filename in the error message string returned from `what()`. Here is a `FileReadError` derived from `FileError`:

```
class FileReadError : public FileError
{
public:
    FileReadError(const string& fileNameIn, int lineNumIn);
    int getLineNum() { return mLineNum; }
private:
    int mLineNum;
};

FileReadError::FileReadError(const string& fileNameIn, int lineNumIn)
: FileError(fileNameIn), mLineNum(lineNumIn)
{
    ostringstream ostr;
    ostr << "Error reading " << fileNameIn << " at line " << lineNumIn;
    setMessage(ostr.str());
}
```

Of course, in order to set the line number properly, you need to modify your `readIntegerFile()` function to track the number of lines read instead of just reading integers directly. Here is a new `readIntegerFile()` function that uses the new exceptions:

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;
    string line;
    int lineNumber = 0;
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw FileOpenError(fileName);
    }
    while (!istr.eof()) {
        // Read one line from the file.
        getline(istr, line);
        lineNumber++;
        // Create a string stream out of the line.
        istringstream lineStream(line);
        // Read the integers one by one and add them to the vector.
        while (lineStream >> temp) {
            dest.push_back(temp);
        }
    }
}
```

```

        if (!lineStream.eof()) {
            // We did not reach the end of the string stream.
            // This means that some error occurred while reading this line.
            // Throw an exception.
            throw FileReadError(fileName, lineNumber);
        }
    }
}

```

Now, code that calls `readIntegerFile()` can use polymorphism to catch exceptions of type `FileError` like this:

```

try {
    readIntegerFile(fileName, myInts);
} catch (const FileError& e) {
    cerr << e.what() << endl;
    return 1;
}

```

There is one trick to writing classes whose objects will be used as exceptions. When a piece of code throws an exception, the object or value thrown is copied. That is, a new object is constructed from the old object by using the copy constructor. It must be copied because the original could go out of scope (and be destroyed and have its memory reclaimed) before the exception is caught, higher up in the stack. Thus, if you write a class whose objects will be thrown as exceptions, you must make those objects copyable. This means that if you have dynamically allocated memory, you must write a destructor, copy constructor, and assignment operator, as described in Chapter 8.

WARNING *Objects thrown as exceptions are always copied by value at least once.*

It is possible for exceptions to be copied more than once, but only if you catch the exception by value instead of by reference.

NOTE *Catch exception objects by reference to avoid unnecessary copying.*

Nested Exceptions

It could happen that during handling of a first exception, a second exceptional situation is triggered which requires a second exception to be thrown. Unfortunately, when you throw the second exception, all information about the first exception that you are currently trying to handle will be lost. The solution provided by C++ for this problem is called *nested exceptions*, which allow you

to nest a caught exception in the context of a new exception. You use `std::throw_with_nested()` to throw an exception with another exception nested inside it. A catch handler for the second exception can use a `dynamic_cast` to get access to the `nested_exception` representing the first exception. The following example demonstrates the use of nested exceptions. This example defines a `MyException` class which derives from `exception` and accepts a string in its constructor.

```
class MyException : public std::exception
{
public:
    MyException(const char* msg) : mMsg(msg) {}
    virtual ~MyException() noexcept {}
    virtual const char* what() const noexcept override
    { return mMsg.c_str(); }
private:
    std::string mMsg;
};
```

When you are handling a first exception and you need to throw a second exception with the first one nested inside it, you need to use the `std::throw_with_nested()` function. The following `doSomething()` function throws a `runtime_error` which is immediately caught in the catch handler. The catch handler writes a message and then uses the `throw_with_nested()` function to throw a second exception that has the first one nested inside it. Note that nesting the exception happens automatically.

```
void doSomething()
{
    try {
        throw std::runtime_error("Throwing a runtime_error exception");
    } catch (const std::runtime_error& e) {
        std::cout << __func__ << " caught a runtime_error" << std::endl;
        std::cout << __func__ << " throwing MyException" << std::endl;
        std::throw_with_nested(
            MyException("MyException with nested runtime_error"));
    }
}
```

The following `main()` function demonstrates how to handle the exception with a nested exception. The code calls the `doSomething()` function and has one catch handler for exceptions of type `MyException`. When it catches such an exception, it writes a message and then uses a `dynamic_cast` to get access to the nested exception. If there is no nested exception inside, the result will be a null pointer. If there is a nested exception inside, the `rethrow_nested()` method on the `nested_exception` is called. This will cause the nested exception to be rethrown which you can then catch in another `try/catch` block.

```
int main()
{
    try {
        doSomething();
    } catch (const MyException& e) {
        std::cout << __func__ << " caught MyException: " << e.what()
        << std::endl;
    }
}
```

```

        const std::nested_exception* pNested =
            dynamic_cast<const std::nested_exception*>(&e);
        if (pNested) {
            try {
                pNested->rethrow_nested();
            } catch (const std::runtime_error& e) {
                // Handle nested exception
                std::cout << " Nested exception: " << e.what()
                    << std::endl;
            }
        }
    }
    return 0;
}

```

The output should be as follows:

```

doSomething caught a runtime_error
doSomething throwing MyException
main caught MyException: MyException with nested runtime_error
    Nested exception: Throwing a runtime_error exception

```

The preceding `main()` function uses a `dynamic_cast` to check for the nested exception. Since you often have to perform this `dynamic_cast` if you want to check for a nested exception, the standard provides a small wrapper called `std::rethrow_if_nested()` that does it for you. This wrapper can be used as follows:

```

int main()
{
    try {
        doSomething();
    } catch (const MyException& e) {
        std::cout << __func__ << " caught MyException: " << e.what()
            << std::endl;
        try {
            std::rethrow_if_nested(e);
        } catch (const std::runtime_error& e) {
            // Handle nested exception
            std::cout << " Nested exception: " << e.what() << std::endl;
        }
    }
    return 0;
}

```

STACK UNWINDING AND CLEANUP

When a piece of code throws an exception, it searches for a catch handler on the stack. This catch handler could be zero or more function calls up the stack of execution. When one is found, the stack is stripped back to the stack level that defines the catch handler by unwinding all intermediate

stack frames. *Stack unwinding* means that the destructors for all locally-scoped names are called and all code remaining in each function past the current point of execution is skipped.

However, in stack unwinding, pointer variables are not freed, and other cleanup is not performed. This behavior can present problems, as the following code demonstrates:

```
void funcOne();
void funcTwo();
int main()
{
    try {
        funcOne();
    } catch (const exception& e) {
        cerr << "Exception caught!" << endl;
        return 1;
    }
    return 0;
}
void funcOne()
{
    string str1;
    string* str2 = new string();
    funcTwo();
    delete str2;
}
void funcTwo()
{
    ifstream istr;
    istr.open("filename");
    throw exception();
    istr.close();
}
```

When `funcTwo()` throws an exception, the closest exception handler is in `main()`. Control then jumps immediately from this line in `funcTwo()`:

```
throw exception();
```

to this line in `main()`:

```
cerr << "Exception caught!" << endl;
```

In `funcTwo()`, control remains at the line that threw the exception, so this subsequent line never gets a chance to run:

```
istr.close();
```

However, luckily for you, the `ifstream` destructor is called because `istr` is a local variable on the stack. The `ifstream` destructor closes the file for you, so there is no resource leak here. If you had dynamically allocated `istr`, it would not be destroyed, and the file would not be closed.

In `funcOne()`, control is at the call to `funcTwo()`, so this subsequent line never gets a chance to run:

```
delete str2;
```

In this case, there really is a memory leak. Stack unwinding does not automatically call `delete` on `str2` for you. However, `str1` is destroyed properly because it is a local variable on the stack. Stack unwinding destroys all local variables correctly.

WARNING *Careless exception handling can lead to memory and resource leaks.*

This is one reason why you should never mix older C models of allocation (even if you are calling `new` so it looks like C++) with modern programming methodologies like exceptions. In C++, this situation should be handled by using stack-based allocations, or if that is not possible, by one of the techniques discussed in the following two sections.

Use Smart Pointers

If stack-based allocation is not possible then use smart pointers. They allow you to write code that automatically prevents memory or resource leaks with exception handling. Smart pointer objects are allocated on the stack and whenever the smart pointer object is destroyed, it frees the underlying resource. Here is an example of the previous `funcOne()` function but using the `unique_ptr` smart pointer:

```
#include <memory>
using namespace std;
void funcOne()
{
    string str1;
    auto str2 = make_unique<string>("hello");
    funcTwo();
}
```

The `str2` pointer of type `string*` will automatically be deleted when you return from `funcOne()` or when an exception is thrown.

NOTE *With smart pointers, you never have to remember to free the underlying resource: the smart pointer destructor does it for you, whether you leave the function via an exception or leave the function normally.*

Catch, Cleanup, and Rethrow

The next technique for avoiding memory and resource leaks is for each function to catch any possible exceptions, perform necessary cleanup work, and rethrow the exception for the function higher up the stack to handle. Here is a revised `funcOne()` with this technique:

```
void funcOne()
{
    string str1;
```

```
string* str2 = new string();
try {
    funcTwo();
} catch (...) {
    delete str2;
    throw; // Rethrow the exception.
}
delete str2;
```

This function wraps the call to `funcTwo()` with an exception handler that performs the cleanup (calls `delete` on `str2`) and then rethrows the exception. The keyword `throw` by itself rethrows whatever exception was caught most recently. Note that the `catch` statement uses the `...` syntax to catch any exception.

This method works fine, but can be messy. In particular, note that there are now two identical lines that call `delete` on `str2`: one to handle the exception and one if the function exits normally.

WARNING *The preferred solution is to use smart pointers instead of the catch, cleanup, and rethrow technique.*

COMMON ERROR-HANDLING ISSUES

Whether or not you use exceptions in your programs is up to you and your colleagues. However, you are strongly encouraged to formalize an error-handling plan for your programs, regardless of your use of exceptions. If you use exceptions, it is generally easier to come up with a unified error-handling scheme, but it is not impossible without exceptions. The most important aspect of a good plan is uniformity of error handling throughout all the modules of the program. Make sure that every programmer on the project understands and follows the error-handling rules.

This section discusses the most common error-handling issues in the context of exceptions, but the issues are also relevant to programs that do not use exceptions.

Memory Allocation Errors

Despite the fact that all the examples so far in this book have ignored the possibility, memory allocation can, and will, fail. However, production code must account for memory allocation failures. C++ provides several different ways to handle memory errors.

The default behaviors of `new` and `new[]` are to throw an exception of type `bad_alloc`, defined in the `<new>` header file, if they cannot allocate memory. Your code could catch these exceptions and handle them appropriately.

It's not realistic to wrap all your calls to `new` and `new[]` with a `try/catch`, but at least you should do so when you are trying to allocate a big block of memory. The following example demonstrates how to catch memory allocation exceptions.

```
try {
    ptr = new int[numInts];
} catch (const bad_alloc& e) {
    cerr << __FILE__ << "(" << __LINE__
        << "): Unable to allocate memory: " << e.what() << endl;
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.
```

Note that this code uses the predefined preprocessor symbols `__FILE__` and `__LINE__` which will be replaced with the name of the file and the current line number. This makes debugging easier.

NOTE *This example prints an error message to `cerr`. This assumes your program is running with a console. In GUI applications, you often don't have a console in which case you need to show the error in a GUI specific way to the user.*

You could, of course, bulk handle many possible new failures with a single `try/catch` block at a higher point in the program, if it will work for your program.

Another consideration is that logging an error might try to allocate memory. If `new` fails, there might not be enough memory left even to log the error message.

Non-Throwing `new`

If you don't like exceptions, you can revert to the old C model in which memory allocation routines return a null pointer if they cannot allocate memory. C++ provides `nothrow` versions of `new` and `new[]`, which return `nullptr` instead of throwing an exception if they fail to allocate memory. This is done by using the syntax `new(nothrow)` instead of `new` as shown in the following example.

```
ptr = new(nothrow) int[numInts];
if (ptr == nullptr) {
    cerr << __FILE__ << "(" << __LINE__
        << "): Unable to allocate memory!" << endl;
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.
```

The syntax is a little strange: you really do write “`nothrow`” as if it's an argument to `new` (which it is).

Customizing Memory Allocation Failure Behavior

C++ allows you to specify a *new handler* callback function. By default, there is no new handler, so `new` and `new[]` just throw `bad_alloc` exceptions. However, if there is a new handler, the memory allocation routine calls the new handler upon memory allocation failure instead of throwing an exception. If the new handler returns, the memory allocation routines attempt to allocate memory again, calling the new handler again if they fail. This cycle could become an infinite loop unless your new handler changes the situation with one of four alternatives. Practically speaking, some of the four options are better than others. Here is the list with commentary:

- **Make more memory available.** One trick to expose space is to allocate a large chunk of memory at program start-up, and then to free it in the new handler. A practical example is when you hit an allocation error and you need to save the user state so no work gets lost. The key is to allocate a block of memory at program start-up large enough to allow a complete document save operation. When the new handler is triggered, you free this block, save the document, restart the application and let it reload the saved document.
- **Throw an exception.** The C++ standard mandates that if you throw an exception from your new handler, it must be a `bad_alloc` exception or an exception derived from `bad_alloc`. For example, when your new handler is triggered, you can throw a `document_recovery_alloc` exception which inherits from `bad_alloc`. You can catch this exception somewhere in your application and trigger the document save operation and restart of the application.
- **Set a different new handler.** Theoretically, you could have a series of new handlers, each of which tries to create memory and sets a different new handler if it fails. However, such a scenario is usually more complicated than useful.
- **Terminate the program.** Your new handler can log an error message and throw an agreed-upon exception such as `PleaseTerminateMe` derived from `bad_alloc`. In your top-level function, for example `main()`, you catch this exception and handle it by **returning** from the top-level function. Never explicitly terminate the program by using `exit()` or `abort()`, only by returning from the top-level function. If there are some memory allocations that can fail but still allow your program to succeed, you can simply set the new handler back to its default of `nullptr` temporarily before calling `new` in those cases.

If you don't do one of these four things in your new handler, any memory allocation failure will cause an infinite loop.

You set the new handler with a call to `set_new_handler()`, declared in the `<new>` header file. `set_new_handler()` completes the trio of C++ functions to set callback functions. The other two are `set_terminate()` and `set_unexpected()`, which are discussed earlier in this chapter. Here is an example of a new handler that logs an error message and throws an exception:

```
class PleaseTerminateMe : public bad_alloc { };
void myNewHandler()
{
    cerr << "Unable to allocate memory." << endl;
    throw PleaseTerminateMe();
}
```

The new handler must take no arguments and return no value. This new handler throws a `PleaseTerminateMe` exception like suggested in the fourth bullet in the preceding list.

You can set the new handler like this:

```
int main()
{
    try {
        // Set the new new_handler and save the old.
        new_handler oldHandler = set_new_handler(myNewHandler);
        // Generate allocation error
        size_t numInts = numeric_limits<int>::max();
        int* ptr = new int[numInts];
        // reset the old new_handler
        set_new_handler(oldHandler);
    } catch (const PleaseTerminateMe&) {
        cerr << __FILE__ << "(" << __LINE__
            << "): Terminating program." << endl;
        return 1;
    }
    return 0;
}
```

Note that `new_handler` is a `typedef` for the type of function pointer that `set_new_handler()` takes.

Errors in Constructors

Before C++ programmers discover exceptions, they are often stymied by error handling and constructors. What if a constructor fails to construct the object properly? Constructors don't have a return value, so the standard pre-exception error-handling mechanism doesn't work. Without exceptions, the best you can do is to set a flag in the object specifying that it is not constructed properly. You can provide a method, with a name like `checkConstructionStatus()`, which returns the value of that flag, and hope that clients remember to call the function on the object after constructing it.

Exceptions provide a much better solution. You can throw an exception from a constructor, even though you can't return a value. With exceptions you can easily tell clients whether or not construction of the object succeeded. However, there is one major problem: if an exception leaves a constructor, the destructor for that object will never be called. Thus, you must be careful to clean up any resources and free any allocated memory in constructors before allowing exceptions to leave the constructor. This problem is the same as in any other function, but it is subtler in constructors because you're accustomed to letting the destructors take care of the memory deallocation and resource freeing.

This section describes a `Matrix` class as an example in which the constructor correctly handles exceptions. Note that this example is using a naked pointer called `mMatrix` to demonstrate the problem. In production quality code, you should avoid using naked pointers, for example by using an STL container. The definition of the `Matrix` class looks as follows:

```
#include <cstddef>
#include "Element.h"
class Matrix
```

```

{
public:
    Matrix(size_t width, size_t height);
    virtual ~Matrix();
private:
    size_t mWidth;
    size_t mHeight;
    Element** mMatrix;
};

```

The preceding class uses the `Element` class, which is kept at a bare minimum for this example:

```

class Element
{
private:
    int mValue;
};

```

The implementation of the `Matrix` class is as follows. Note that the first call to `new` is not protected with a `try/catch` block. It doesn't matter if the first `new` throws an exception because the constructor hasn't allocated anything else yet that needs freeing. If any of the subsequent `new` calls throw exceptions, though, the constructor must clean up all of the memory already allocated. However, it doesn't know what exceptions the `Element` constructors themselves might throw, so it catches any exception via `...` and translates them into a `bad_alloc` exception. It is also important to have `index i` outside the `try` block because this index is needed during cleanup in the `catch` block.

```

Matrix::Matrix(size_t width, size_t height)
    : mWidth(width), mHeight(height), mMatrix(nullptr)
{
    mMatrix = new Element*[width];
    size_t i = 0;
    try {
        for (i = 0; i < width; ++i)
            mMatrix[i] = new Element[height];
    } catch (...) {
        cout << "Exception caught in constructor, cleaning up..." << endl;
        // Clean up any memory we already allocated, because the destructor
        // will never be called. The upper bound of the for loop is the
        // index of the last element in the mMatrix array that we tried
        // to allocate (the one that failed). All indices before that
        // one store pointers to allocated memory that must be freed.
        for (size_t j = 0; j < i; j++) {
            delete [] mMatrix[j];
        }
        delete [] mMatrix;
        mMatrix = nullptr;
        // Translate any exception to bad_alloc.
        throw bad_alloc();
    }
}
Matrix::~Matrix()
{
    for (size_t i = 0; i < mWidth; ++i)

```

```

        delete [] mMMatrix[i];
        delete [] mMMatrix;
        mMMatrix = nullptr;
    }
}

```

WARNING Remember, if an exception leaves a constructor, the destructor for that object will never be called!

You might be wondering what happens when you add inheritance into the mix.

NOTE C++ guarantees that it will run the destructor for any fully constructed “subobjects.” Therefore, any constructor that completes without an exception will cause the corresponding destructor to be run.

Base class constructors run before derived class constructors. If a derived class constructor throws an exception, C++ will execute the destructor of the fully constructed base class.

Function-Try-Blocks for Constructors

The exception mechanism as discussed up to now in this chapter is perfect to handle exceptions within functions. However, how should you handle exceptions thrown from inside a ctor-initializer of a constructor? This section explains a feature called *function-try-blocks*, which are capable of catching those exceptions. Function-try-blocks work for normal functions as well as for constructors. This section focuses on the use with constructors. Most C++ programmers, even experienced C++ programmers don’t know the existence of this feature, even though it was introduced more than a decade ago.

The following piece of pseudo code shows the basic syntax for a function-try-block for a constructor:

```

 MyClass::MyClass()
 try
 : <ctor-initializer>
{
    /* ... constructor body ... */
}
 catch (const exception& e)
{
    /* ... */
}

```

As you can see, the `try` keyword should be right before the start of the `ctor-initializer`. The `catch` statements should be after the closing brace for the constructor, actually putting them outside the constructor body. There are a number of restrictions and guidelines that you should keep in mind when using function-try-blocks with constructors:

- The catch statements will catch any exception either thrown directly or indirectly by the ctor-initializer or by the constructor body.
- The catch statements have to rethrow the current exception or throw a new exception. If a catch statement doesn't do this, the runtime will automatically rethrow the current exception.
- The catch statements can access arguments passed to the constructor.
- When a catch statement catches an exception in a function-try-block, all fully constructed base classes and members of the object will be destroyed before execution of the catch statement starts.
- Inside catch statements you should not access member variables that are objects because these will be destroyed prior to executing the catch statements (see previous bullet). However, if your object contains non-class data members, for example naked pointers, you can access them if they have been initialized before the exception got thrown. If you have such naked resources, you will have to take care of them by freeing them in the catch statements.
- The catch statements in a function-try-block cannot use the `return` keyword to return a value from the function enclosed by it. This is not relevant for constructors because they do not return anything.

Based on this list of limitations, function-try-blocks for constructors are useful only in a limited number of situations:

- To convert an exception thrown by the ctor-initializer to another exception.
- To log a message to a log file.
- To free naked resources that have been allocated in the ctor-initializer prior to the exception being thrown.

Let's see how to use function-try-blocks with an example. The following code defines a class called `SubObject`. It has only one constructor, which throws an exception of type `runtime_error`.

```
class SubObject
{
public:
    SubObject(int i);
};

SubObject::SubObject(int i)
{
    throw std::runtime_error("Exception by SubObject ctor");
}
```

The `MyClass` class has a member variable of type `SubObject`:

```
class MyClass
{
public:
    MyClass();
private:
    SubObject mSubObject;
};
```

The `SubObject` class does not have a default constructor. This means that you need to initialize `mSubObject` in the `MyClass` ctor-initializer. The constructor of `MyClass` uses a function-try-block to catch exceptions thrown in its ctor-initializer:

```
MyClass::MyClass()
try
    : mSubObject(42)
{
    /* ... constructor body ... */
}
catch (const std::exception& e)
{
    cout << "function-try-block caught: '" << e.what() << "' " << endl;
}
```

Remember that `catch` statements in a function-try-block for a constructor have to either rethrow the current exception or throw a new exception. The preceding `catch` statement does not throw anything, so the C++ runtime will automatically rethrow the current exception. Following is a simple function that uses the preceding class:

```
int main()
{
    try {
        MyClass m;
    } catch (const std::exception& e) {
        cout << "main() caught: '" << e.what() << "' " << endl;
    }
    return 0;
}
```

The output of the preceding example is as follows:

```
function-try-block caught: 'Exception by SubObject ctor'
main() caught: 'Exception by SubObject ctor'
```

Function-try-blocks are not limited to constructors. They can be used with ordinary functions as well. However, for normal functions, there is no useful reason to use function-try-blocks because they can just as easily be converted to a simple `try/catch` block inside the function body. One notable difference when using a function-try-block on a normal function compared to a constructor is that rethrowing the current exception or throwing a new exception in the `catch` statements is not required and the C++ runtime will not automatically rethrow the exception.

Errors in Destructors

You should handle all error conditions that arise in destructors in the destructors themselves. You should not let any exceptions be thrown from destructors, for three reasons:

1. Destructors can run while there is another pending exception, in the process of stack unwinding. If you throw an exception from the destructor in the middle of stack unwinding, the C++ runtime will call `std::terminate()` to terminate the application. For the brave

and curious, C++ does provide the ability to determine, in a destructor, whether you are executing as a result of a normal function exit or delete call, or because of stack unwinding. The function `uncaught_exception()`, declared in the `<exception>` header file, returns `true` if there is an uncaught exception and you are in the middle of stack unwinding. Otherwise, it returns `false`. However, this approach is messy and should be avoided.

2. What action would clients take? Clients don't call destructors explicitly: they call `delete`, which calls the destructor. If you throw an exception from the destructor, what is a client supposed to do? It can't call `delete` on the object again, and it shouldn't call the destructor explicitly. There is no reasonable action the client can take, so there is no reason to burden that code with exception handling.
3. The destructor is your one chance to free memory and resources used in the object. If you waste your chance by exiting the function early due to an exception, you will never be able to go back and free the memory or resources.

WARNING *Be careful to catch in a destructor any exceptions that can be thrown by calls you make from the destructor.*

PUTTING IT ALL TOGETHER

Now that you've learned about error handling and exceptions, let's see it all coming together in a bigger example, a `GameBoard` class. This `GameBoard` class is based on the `GameBoard` class from Chapter 11. The implementation in Chapter 11 using a vector of vectors is the recommended implementation because even when an exception is thrown, the code is not leaking any memory due to the use of STL containers. To be able to demonstrate handling memory allocation errors, the version below is adapted to use a naked pointer `GamePiece** mCells`. First, here is the definition of the class without any exceptions.

```
class GamePiece {};

class GameBoard
{
public:
    // general-purpose GameBoard allows user to specify its dimensions
    explicit GameBoard(size_t inWidth = kDefaultWidth,
                       size_t inHeight = kDefaultHeight);
    GameBoard(const GameBoard& src); // Copy constructor
    virtual ~GameBoard();
    GameBoard& operator=(const GameBoard& rhs); // Assignment operator
    void setPieceAt(size_t x, size_t y, const GamePiece& inPiece);
    GamePiece& getPieceAt(size_t x, size_t y);
    const GamePiece& getPieceAt(size_t x, size_t y) const;
    size_t getHeight() const { return mHeight; }
    size_t getWidth() const { return mWidth; }
    static const size_t kDefaultWidth = 100;
    static const size_t kDefaultHeight = 100;
private:
    void copyFrom(const GameBoard& src);
```

```

        void freeMemory();
        // Objects dynamically allocate space for the game pieces.
        GamePiece** mCells;
        size_t mWidth, mHeight;
    };
}

```

And here is the implementation without any exceptions:

```

GameBoard::GameBoard(size_t inWidth, size_t inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new GamePiece* [mWidth];
    for (size_t i = 0; i < mWidth; i++) {
        mCells[i] = new GamePiece[mHeight];
    }
}

GameBoard::GameBoard(const GameBoard& src)
{
    copyFrom(src);
}

GameBoard::~GameBoard()
{
    // Free the old memory
    freeMemory();
}

void GameBoard::copyFrom(const GameBoard& src)
{
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    mCells = new GamePiece* [mWidth];
    for (size_t i = 0; i < mWidth; i++) {
        mCells[i] = new GamePiece[mHeight];
    }
    for (size_t i = 0; i < mWidth; i++) {
        for (size_t j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

void GameBoard::freeMemory()
{
    for (size_t i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }
    delete[] mCells;
    mCells = nullptr;
    mWidth = 0;
    mHeight = 0;
}

```

```

GameBoard& GameBoard::operator=(const GameBoard& rhs)
{
    // Check for self-assignment
    if (this == &rhs) {
        return *this;
    }
    // Free the old memory
    freeMemory();
    // Copy the new memory
    copyFrom(rhs);
    return *this;
}
void GameBoard::setPieceAt(size_t x, size_t y, const GamePiece& inElem)
{
    mCells[x][y] = inElem;
}
GamePiece& GameBoard::getPieceAt(size_t x, size_t y)
{
    return mCells[x][y];
}
const GamePiece& GameBoard::getPieceAt(size_t x, size_t y) const
{
    return mCells[x][y];
}

```

Now, let's retrofit the preceding class to include error handling and exceptions. The constructors, `operator=` and `copyFrom()` can all throw `bad_alloc` because they perform memory allocation. The destructor, `getHeight()`, and `getWidth()` throw no exceptions. `setPieceAt()` and `getPieceAt()` throw `out_of_range` if the caller supplies an invalid coordinate. Here is the retrofitted class definition:

```

class GamePiece {};

class GameBoard
{
public:
    explicit GameBoard(size_t inWidth = kDefaultWidth,
                       size_t inHeight = kDefaultHeight);
    GameBoard(const GameBoard& src);
    virtual ~GameBoard() noexcept;
    GameBoard& operator=(const GameBoard& rhs);
    void setPieceAt(size_t x, size_t y, const GamePiece& inPiece);
    GamePiece& getPieceAt(size_t x, size_t y);
    const GamePiece& getPieceAt(size_t x, size_t y) const;
    size_t getHeight() const noexcept { return mHeight; }
    size_t getWidth() const noexcept { return mWidth; }
    static const size_t kDefaultWidth = 100;
    static const size_t kDefaultHeight = 100;
private:
    void copyFrom(const GameBoard& src);
    void freeMemory();
    void allocateMemory();
    GamePiece** mCells;
    size_t mWidth, mHeight;
};

```

Here are the implementations with exception handling. `getPieceAt()` is not shown because it is similar to `setPieceAt()`. The operator`=` is also not shown because it did not change.

```
GameBoard::GameBoard(size_t inWidth, size_t inHeight) :
mWidth(inWidth), mHeight(inHeight)
{
    allocateMemory();
}
void GameBoard::allocateMemory()
{
    size_t i = 0;
    mCells = new GamePiece*[mWidth];
    try {
        for (i = 0; i < mWidth; i++) {
            mCells[i] = new GamePiece[mHeight];
        }
    } catch (...) {
        // allocateMemory() is called from the constructor and from the
        // copy constructor, so we need to clean up any memory we already
        // allocated, because the destructor will never get called.
        // The upper bound of the for loop is the index of the last
        // element in the mCells array that we tried to allocate
        // (the one that failed). All indices before that one store
        // pointers to allocated memory that must be freed.
        for (size_t j = 0; j < i; j++) {
            delete[] mCells[j];
        }
        // allocateMemory() is called from the constructor, from the
        // copy constructor, and indirectly called from operator=. In the
        // latter case the object was already constructed, so the
        // destructor will be called at some point.
        // So, set mCells, mWidth, and mHeight to values that will allow
        // the destructor to run without harming anything.
        delete[] mCells;
        mCells = nullptr;
        mWidth = 0;
        mHeight = 0;
        // translate any exception to bad_alloc
        throw bad_alloc();
    }
}
void GameBoard::freeMemory()
{
    for (size_t i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }
    delete[] mCells;
    mCells = nullptr;
    mWidth = 0;
    mHeight = 0;
}
GameBoard::GameBoard(const GameBoard& src)
{
    copyFrom(src);
}
```

```
GameBoard::~GameBoard() noexcept
{
    // free the old memory
    freeMemory();
}
void GameBoard::copyFrom(const GameBoard& src)
{
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    allocateMemory();
    // Copy data.
    for (size_t i = 0; i < mWidth; i++) {
        for (size_t j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}
void GameBoard::setPieceAt(size_t x, size_t y, const GamePiece& inElem)
{
    // Check for out of range arguments
    if (x >= mWidth)
        throw out_of_range("GameBoard::setPieceAt: x-coordinate beyond width");
    if (y >= mHeight)
        throw out_of_range("GameBoard::setPieceAt: y-coordinate beyond height");
    mCells[x][y] = inElem;
}
```

SUMMARY

This chapter described the issues related to error handling in C++ programs, and emphasized that you must design and code your programs with an error-handling plan. By reading this chapter, you learned the details of C++ exceptions syntax and behavior. The chapter also covered some of the areas in which error handling plays a large role, including I/O streams, memory allocation, constructors, and destructors. Finally, you saw an example of error handling in a `GameBoard` class.

14

Overloading C++ Operators

WHAT'S IN THIS CHAPTER?

- Explaining operator overloading
- Rationale for overloading operators
- Limitations, caveats, and choices in operator overloading
- Summary of operators you can, cannot, and should not overload
- How to overload unary plus, unary minus, increment, and decrement
- How to overload the I/O streams operators (`operator<<` and `operator>>`)
- How to overload the subscripting (array index) operator
- How to overload the function call operator
- How to overload the dereferencing operators (`*` and `->`)
- How to write conversion operators
- How to overload the memory allocation and deallocation operators

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

C++ allows you to redefine the meanings of operators, such as `+`, `-`, and `=`, for your classes. Many object-oriented languages do not provide this capability, so you might be tempted to disregard its usefulness in C++. However, it can be beneficial for making your classes behave

similarly to built-in types such as `ints` and `doubles`. It is even possible to write classes that look like arrays, functions, or pointers.

Chapters 5 and 6 introduce object-oriented design and operator overloading, respectively. Chapters 7 and 8 present the syntax details for objects and for basic operator overloading. This chapter picks up operator overloading where Chapter 8 left off.

OVERVIEW OF OPERATOR OVERLOADING

As Chapter 1 explains, operators in C++ are symbols such as `+`, `<`, `*`, and `<<`. They work on built-in types such as `int` and `double` to allow you to perform arithmetic, logical, and other operations. There are also operators such as `->` and `*` that allow you to dereference pointers. The concept of operators in C++ is broad, and even includes `[]` (array index), `()` (function call), casting, and the memory allocation and deallocation routines. Operator overloading allows you to change the behavior of language operators for your classes. However, this capability comes with rules, limitations, and choices.

Why Overload Operators?

Before learning how to overload operators, you probably want to know why you would ever want to do so. The reasons vary for the different operators, but the general guiding principle is to make your classes behave like built-in types. The closer your classes are to built-in types, the easier they will be for clients to use. For example, if you want to write a class to represent fractions, it's quite helpful to have the ability to define what `+`, `-`, `*`, and `/` mean when applied to objects of that class.

The second reason to overload operators is to gain greater control over the behavior in your program. For example, you can overload memory allocation and deallocation routines for your classes to specify exactly how memory should be distributed and reclaimed for each new object.

It's important to emphasize that operator overloading doesn't necessarily make things easier for you as the class developer; its main purpose is to make things easier for clients of the class.

Limitations to Operator Overloading

Here is a list of things you cannot do when you overload operators:

- You cannot add new operator symbols. You can only redefine the meanings of operators already in the language. The table in the “Summary of Overloadable Operators” section lists all of the operators that you can overload.
- There are a few operators that you cannot overload, such as `.` (member access in an object), `::` (scope resolution operator), `sizeof`, `?:` (the conditional operator), and a few others. The table lists all the operators that you *can* overload. The operators that you *can't* overload are usually not those you would care to overload anyway, so you shouldn't find this restriction limiting.
- The *arity* describes the number of arguments, or *operands*, associated with the operator. You can only change the arity for the function call, new, and delete operators. For all other operators you cannot change the arity. Unary operators, such as `++`, work on only one

operand. Binary operators, such as `/`, work on two operands. There is only one ternary operator: `? :`. The main place where this limitation might bother you is when overloading `[]` (array brackets), discussed later in this chapter.

- You cannot change the *precedence* or *associativity* of the operator. These rules determine in which order operators are evaluated in a statement. Again, this constraint shouldn't be cause for concern in most programs because there are rarely benefits to changing the order of evaluation.
- You cannot redefine operators for built-in types. The operator must be a method in a class, or at least one of the arguments to a global overloaded operator function must be a user-defined type (e.g., a class). This means that you can't do something ridiculous, such as redefine `+` for `ints` to mean subtraction (though you could do so for your classes). The one exception to this rule is the memory allocation and deallocation routines; you can replace the global routines for all memory allocations in your program.

Some of the operators already mean two different things. For example, the `-` operator can be used as a binary operator, as in `x = y - z`; or as a unary operator, as in `x = -y`. The `*` operator can be used for multiplication or for dereferencing a pointer. The `<<` operator is the insertion operator or the left-shift operator, depending on the context. You can overload both meanings of operators with dual meanings.

Choices in Operator Overloading

When you overload an operator, you write a function or method with the name `operatorX`, where `X` is the symbol for some operator, and with optional whitespace between `operator` and `X`. For example, Chapter 8 declares `operator+` for `SpreadsheetCell` objects like this:

```
friend SpreadsheetCell operator+(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs);
```

The following sections describe several choices involved in each overloaded operator function or method you write.

Method or Global Function

First, you must decide whether your operator should be a method of your class or a global function (usually a `friend` of the class). How do you choose? First, you need to understand the difference between these two choices. When the operator is a method of a class, the left-hand side of the operator expression must always be an object of that class. If you write a global function, the left-hand side can be an object of a different type.

There are three different types of operators:

- **Operators that must be methods:** The C++ language requires some operators to be methods of a class because they don't make sense outside of a class. For example, `operator=` is tied so closely to the class that it can't exist anywhere else. The table in the “Summary of Overloadable Operators” section lists those operators that must be methods. Most operators do not impose this requirement.
- **Operators that must be global functions:** Whenever you need to allow the left-hand side of the operator to be a variable of a different type than your class, you must make the operator

a global function. This rule applies specifically to `operator<<` and `operator>>`, where the left-hand side is the `iostream` object, not an object of your class. Additionally, commutative operators like binary `+` and `-` should allow variables that are not objects of your class on the left-hand side. Chapter 8 mentions this problem.

- **Operators that can be either methods or global functions:** There is some disagreement in the C++ community on whether it's better to write methods or global functions to overload operators. However, I recommend the following rule: Make every operator a method unless you must make it a global function, as described previously. One major advantage to this rule is that methods can be `virtual`, but `friend` functions cannot. Therefore, when you plan to write overloaded operators in an inheritance tree, you should make them methods if possible.

When you write an overloaded operator as a method, you should mark it `const` if it doesn't change the object. That way, it can be called on `const` objects.

Choosing Argument Types

You are somewhat limited in your choice of argument types because as stated earlier for most operators you cannot change the number of arguments. For example, `operator/` must always have two arguments if it is a global function; one argument if it's a method. The compiler issues an error if it differs from this standard. In this sense, the operator functions are different from normal functions, which you can overload with any number of parameters. Additionally, although you can write the operator for whichever types you want, the choice is usually constrained by the class for which you are writing the operator. For example, if you want to implement addition for class `T`, you wouldn't write an `operator+` that takes two `string`s! The real choice arises when you try to determine whether to take parameters by value or by reference, and whether or not to make them `const`.

The choice of value vs. reference is easy: you should take every non-primitive parameter type by reference. As Chapters 8 and 10 explain, never pass objects by value if you can pass-by-reference instead.

The `const` decision is also trivial: mark every parameter `const` unless you actually modify it. The table in the “Summary of Overloadable Operators” section shows sample prototypes for each operator, with the arguments marked `const` and reference as appropriate.

Choosing Return Types

C++ doesn't determine overload resolution based on return type. Thus, you can specify any return type you want when you write overloaded operators. However, just because you *can* do something doesn't mean you *should* do it. This flexibility implies that you could write confusing code in which comparison operators return pointers, and arithmetic operators return `bool`s. However, you shouldn't do that. Instead, you should write your overloaded operators such that they return the same types as the operators do for the built-in types. If you write a comparison operator, return a `bool`. If you write an arithmetic operator, return an object representing the result of the arithmetic. Sometimes the return type is not obvious at first. For example, as Chapter 7 mentions, `operator=` should return a reference to the object on which it's called in order to support nested assignments. Other operators have similarly tricky return types, all of which are summarized in the table in the “Summary of Overloadable Operators” section.

The same choices of reference and `const` apply to return types as well. However, for return values, the choices are more difficult. The general rule for value or reference is to return a reference if you can; otherwise, return a value. How do you know when you can return a reference? This choice applies only to operators that return objects: the choice is moot for the comparison operators that return `bool`; the conversion operators that have no return type; and the function call operator, which may return any type you want. If your operator constructs a new object, then you must return that new object by value. If it does not construct a new object, you can return a reference to the object on which the operator is called, or one of its arguments. The table in the “Summary of Overloadable Operators” section shows examples.

A return value that can be modified as an *lvalue* (the left-hand side of an assignment expression) must be `non-const`. Otherwise, it should be `const`. More operators than you might think at first require that you return *lvalues*, including all of the assignment operators (`operator=`, `operator+=`, `operator-=`, etc.).

Choosing Behavior

You can provide whichever implementation you want in an overloaded operator. For example, you could write an `operator+` that launches a game of Scrabble. However, as Chapter 6 describes, you should generally constrain your implementations to provide behaviors that clients expect. Write `operator+` so that it performs addition, or something like addition, such as string concatenation. This chapter explains how you *should* implement your overloaded operators. In exceptional circumstances, you might want to differ from these recommendations; but, in general, you should follow the standard patterns.

Operators You Shouldn’t Overload

Some operators should not be overloaded, even though it is permitted. Specifically, the address-of operator (`operator&`) is not particularly useful to overload, and leads to confusion if you do because you are changing fundamental language behavior (taking addresses of variables) in potentially unexpected ways. The entire STL, which uses operator overloading extensively, never overloads the address-of operator.

Additionally, you should avoid overloading the binary Boolean operators `operator&&` and `operator||` because you lose C++’s short-circuit evaluation rules.

Finally, you should not overload the comma operator (`operator,`). Yes, you read that correctly: there really is a comma operator in C++. It’s also called the *sequencing operator*, and is used to separate two expressions in a single statement, while guaranteeing that they are evaluated left to right. There is rarely (if ever) a good reason to overload this operator.

Summary of Overloadable Operators

The following table lists the operators that you can overload, specifies whether they should be methods of the class or global `friend` functions, summarizes when you should (or should not) overload them, and provides sample prototypes showing the proper return values.

This table should be a useful reference in the future when you want to write an overloaded operator. You’re bound to forget which return type you should use, and whether or not the function should be a method.

In this table, `T` is the name of the class for which the overloaded operator is written, and `E` is a different type. Note that the sample prototypes given are not exhaustive; often there are other combinations of `T` and `E` possible for a given operator.

OPERATOR	NAME OR CATEGORY	METHOD OR GLOBAL FRIEND FUNCTION	WHEN TO OVERLOAD	SAMPLE PROTOTYPE
<code>operator+</code> <code>operator-</code> <code>operator*</code> <code>operator/</code> <code>operator%</code>	Binary arithmetic	Global friend function recommended	Whenever you want to provide these operations for your class	<code>friend T operator+(const T&, const T&);</code> <code>friend T operator+(const T&, const E&);</code>
<code>operator-</code> <code>operator+</code> <code>operator~</code>	Unary arithmetic and bitwise operators	Method recommended	Whenever you want to provide these operations for your class	<code>T operator-() const;</code>
<code>operator++</code> <code>operator--</code>	Pre-increment and pre-decrement	Method recommended	Whenever you overload <code>+=</code> and <code>-=</code>	<code>T& operator++();</code>
<code>operator++</code> <code>operator--</code>	Post-increment and post-decrement	Method recommended	Whenever you overload <code>+=</code> and <code>-=</code>	<code>T operator++(int);</code>
<code>operator=</code>	Assignment operator	Method required	Whenever your class has dynamically allocated memory or resources, or members that are references	<code>T& operator=(const T&);</code>
<code>operator+=</code> <code>operator-=</code> <code>operator*=</code> <code>operator/=</code> <code>operator%=></code>	Shorthand arithmetic operator assignments	Method recommended	Whenever you overload the binary arithmetic operators and your class is not designed to be immutable	<code>T& operator+=(const T&);</code> <code>T& operator+=(const E&);</code>

<code>operator<<</code> <code>operator>></code> <code>operator&</code> <code>operator </code> <code>operator^</code>	Binary bitwise operators	Global friend function recommended	Whenever you want to provide these operations	<code>friend T operator<< (const T&, const T&);</code> <code>friend T operator<< (const T&, const E&);</code>
<code>operator<=</code> <code>operator>=</code> <code>operator&=</code> <code>operator =</code> <code>operator^=</code>	Shorthand bitwise operator assignments	Method recommended	Whenever you overload the binary bitwise operators and your class is not designed to be immutable	<code>T& operator<= (const T&);</code> <code>T& operator<= (const E&);</code>
<code>operator<</code> <code>operator></code> <code>operator<=</code> <code>operator>=</code> <code>operator==</code> <code>operator!=</code>	Binary comparison operators	Global friend function recommended	Whenever you want to provide these operations	<code>friend bool operator< (const T&, const T&);</code> <code>friend bool operator< (const T&, const E&);</code>
<code>operator<<</code> <code>operator>></code>	I/O stream operators (insertion and extraction)	Global friend function required	Whenever you want to provide these operations	<code>friend ostream& operator<< (ostream&, const T&);</code> <code>friend istream& operator>> (istream&, T&);</code>
<code>operator!</code>	Boolean negation operator	Member function recommended	Rarely; use <code>bool</code> or <code>void*</code> conversion instead	<code>bool operator! () const;</code>
<code>operator&&</code> <code>operator </code>	Binary Boolean operators	Global friend function recommended	Rarely	<code>friend bool operator&& (const T& lhs, const T& rhs);</code>
<code>operator[]</code>	Subscripting (array index) operator	Method required	When you want to support subscripting	<code>E& operator[] (int);</code> <code>const E& operator[] (int) const;</code>

continues

(continued)

OPERATOR	NAME OR CATEGORY	METHOD OR GLOBAL FRIEND FUNCTION	WHEN TO OVERLOAD	SAMPLE PROTOTYPE
operator()	Function call operator	Method required	When you want objects to behave like function pointers	Return type and arguments can vary; see examples in this chapter
operator type()	Conversion, or cast, operators (separate operator for each type)	Method required	When you want to provide conversions from your class to other types	operator type() const;
operator new operator new[]	Memory allocation routines	Method recommended	When you want to control memory allocation for your classes (rarely)	void* operator new(size_t size); void* operator new[] (size_t size);
operator delete operator delete[]	Memory deallocation routines	Method recommended	Whenever you overload the memory allocation routines	void operator delete(void* ptr) noexcept; void operator delete[] (void* ptr) noexcept;
operator* operator->	Dereferencing operators	Method recommended for operator* Method required for operator->	Useful for smart pointers	E& operator*() const; E* operator->() const;
operator&	Address-of operator	N/A	Never	N/A
operator->*	Dereference pointer-to-member	N/A	Never	N/A
operator,	Comma operator	N/A	Never	N/A

Rvalue References

Chapter 10 discusses *rvalue references*, written as `&&` instead of the normal lvalue references, `&`. They are demonstrated in Chapter 10 by defining *move assignment operators*, which are used by the compiler in cases where the second object is a temporary object that will be destroyed after the assignment. The normal assignment operator from the preceding table has the following prototype:

```
T& operator=(const T&);
```

The move assignment operator has almost the same prototype, but uses an rvalue reference. It will modify the argument so it cannot be passed as `const`. Details are explained in Chapter 10:

```
T& operator=(T&&);
```

The preceding table does not include sample prototypes with rvalue reference semantics. However, for most operators it can make sense to write both a version using normal lvalue references and a version using rvalue references. Whether it makes sense depends on implementation details of your class. The `operator=` is one example from Chapter 10. Another example is `operator+` to prevent unnecessary memory allocations. The `std::string` class from the STL, for example, implements an `operator+` using rvalue references as follows (simplified):

```
string operator+(string&& lhs, string&& rhs);
```

The implementation of this operator reuses memory of one of the arguments because they are being passed as rvalue references, meaning both are temporary objects that will be destroyed when this `operator+` is finished. The implementation of the preceding `operator+` has the following effect depending on the size and the capacity of both operands:

```
return std::move(lhs.append(rhs));
```

or

```
return std::move(rhs.insert(0, lhs));
```

In fact, `std::string` defines several overloaded `operator+` operators with different combinations of lvalue references and rvalue references. The following is a list of all `operator+` operators for `std::string` accepting two `strings` as arguments (simplified):

```
string operator+(const string& lhs, const string& rhs);
string operator+(string&& lhs, const string& rhs);
string operator+(const string& lhs, string&& rhs);
string operator+(string&& lhs, string&& rhs);
```

Reusing memory of one of the rvalue reference arguments is implemented in the same way as it is explained for move assignment operators in Chapter 10.

Relational Operators

There is a handy `<utility>` header file included with the C++ Standard Library. It contains quite a few helper functions and classes. It also contains the following set of function templates for relational operators in the `std::rel_ops` namespace:

```
template<class T> bool operator!=(const T& a, const T& b); // Needs operator==
template<class T> bool operator>(const T& a, const T& b); // Needs operator<
template<class T> bool operator<=(const T& a, const T& b); // Needs operator<
template<class T> bool operator>=(const T& a, const T& b); // Needs operator<
```

These function templates define the operators `!=`, `>`, `<=`, and `>=` in terms of the `==` and `<` operators for any class. If you implement `operator==` and `operator<` in your class, you get the other relational operators for free with these templates. You can make these available for your class by simply adding a `#include <utility>` and adding the following `using` declarations:

```
using std::rel_ops::operator!=;
using std::rel_ops::operator>;
using std::rel_ops::operator<=;
using std::rel_ops::operator>=;
```

OVERLOADING THE ARITHMETIC OPERATORS

Chapter 8 shows how to write the binary arithmetic operators and the shorthand arithmetic assignment operators, but it does not cover how to overload the other arithmetic operators.

Overloading Unary Minus and Unary Plus

C++ has several unary arithmetic operators. Two of these are unary minus and unary plus. Here is an example of these operators using `ints`:

```
int i, j = 4;
i = -j;      // Unary minus
i = +i;      // Unary plus
j = +(-i); // Apply unary plus to the result of applying unary minus to i.
j = -(-i); // Apply unary minus to the result of applying unary plus to i.
```

Unary minus negates the operand, while unary plus returns the operand directly. Note that you can apply unary plus or unary minus to the result of unary plus or unary minus. These operators don't change the object on which they are called so you should make them `const`.

Here is an example of a unary `operator-` as a member function for a `SpreadsheetCell` class. Unary plus is usually an identity operation, so this class doesn't overload it:

```
SpreadsheetCell SpreadsheetCell::operator-() const
{
    SpreadsheetCell newCell(*this);
    newCell.set(-mValue); // call set to update mValue and mString
    return newCell;
}
```

`operator-` doesn't change the operand, so this method must construct a new `SpreadsheetCell` with the negated value, and return a copy of it. Thus, it can't return a reference.

Overloading Increment and Decrement

There are four ways to add 1 to a variable:

```
i = i + 1;
i += 1;
++i;
i++;
```

The last two are called the *increment* operators. The first form is *prefix increment*, which adds 1 to the variable, then returns the newly incremented value for use in the rest of the expression. The

second form is *postfix increment*, which returns the old (non-incremented) value for use in the rest of the expression. The decrement operators work similarly.

The two possible meanings for `operator++` and `operator--` (prefix and postfix) present a problem when you want to overload them. When you write an overloaded `operator++`, for example, how do you specify whether you are overloading the prefix or the postfix version? C++ introduced a hack to allow you to make this distinction: the prefix versions of `operator++` and `operator--` take no arguments, while the postfix versions take one unused argument of type `int`.

The prototypes of these overloaded operators for the `SpreadsheetCell` class look like this:

```
SpreadsheetCell& operator++(); // Prefix
SpreadsheetCell operator++(int); // Postfix
SpreadsheetCell& operator--(); // Prefix
SpreadsheetCell operator--(int); // Postfix
```

The return value in the prefix forms is the same as the end value of the operand, so prefix increment and decrement can return a reference to the object on which they are called. The postfix versions of increment and decrement, however, return values that are different from the end values of the operands, so they cannot return references.

Here are the implementations for `operator++`:

```
SpreadsheetCell& SpreadsheetCell::operator++()
{
    set(mValue + 1);
    return *this;
}
SpreadsheetCell SpreadsheetCell::operator++(int)
{
    SpreadsheetCell oldCell(*this); // Save the current value before incrementing
    set(mValue + 1); // Increment
    return oldCell; // Return the old value.
}
```

The implementations for `operator--` are almost identical. Now you can increment and decrement `SpreadsheetCell` objects to your heart's content:

```
SpreadsheetCell c1(4);
SpreadsheetCell c2(4);
c1++;
++c2;
```

Increment and decrement also work on pointers. When you write classes that are smart pointers or iterators, you can overload `operator++` and `operator--` to provide pointer incrementing and decrementing.

OVERLOADING THE BITWISE AND BINARY LOGICAL OPERATORS

The bitwise operators are similar to the arithmetic operators, and the bitwise shorthand assignment operators are similar to the arithmetic shorthand assignment operators. However, they are significantly less common, so no examples are shown here. The table in the “Summary

of “Overloadable Operators” section shows sample prototypes, so you should be able to implement them easily if the need ever arises.

The logical operators are trickier. It’s not recommended to overload `&&` and `||`. These operators don’t really apply to individual types: they aggregate results of Boolean expressions. Additionally, you lose the short-circuit evaluation, because both the left-hand side and the right-hand side have to be evaluated before they can be bound to the parameters of your overloaded operator `&&` and `||`. Thus, it rarely makes sense to overload them for specific types.

OVERLOADING THE INSERTION AND EXTRACTION OPERATORS

In C++, you use operators not only for arithmetic operations, but also for reading from and writing to streams. For example, when you write `ints` and `strings` to `cout` you use the insertion operator `<<`:

```
int number = 10;
cout << "The number is " << number << endl;
```

When you read from streams you use the extraction operator `>>`:

```
int number;
string str;
cin >> number >> str;
```

You can write insertion and extraction operators that work on your classes as well, so that you can read and write them like this:

```
SpreadsheetCell myCell, anotherCell, aThirdCell;
cin >> myCell >> anotherCell >> aThirdCell;
cout << myCell << " " << anotherCell << " " << aThirdCell << endl;
```

Before you write the insertion and extraction operators, you need to decide how you want to stream your class out and how you want to read it in. In this example, the `SpreadsheetCells` will read and write `strings`.

The object on the left of an extraction or insertion operator is the `istream` or `ostream` (such as `cin` or `cout`), not a `SpreadsheetCell` object. Because you can’t add a method to the `istream` or `ostream` classes, you must write the extraction and insertion operators as global `friend` functions of the `SpreadsheetCell` class. The declaration of these functions looks like this:

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend std::ostream& operator<<(std::ostream& ostr,
                                         const SpreadsheetCell& cell);
    friend std::istream& operator>>(std::istream& istr,
                                         SpreadsheetCell& cell);
    // Omitted for brevity
};
```

By making the insertion operator take a reference to an `ostream` as its first parameter, you allow it to be used for file output streams, string output streams, `cout`, `cerr`, and `clog`. See Chapter 12 for details. Similarly, by making the extraction operator take a reference to an `istream`, you make it work on file input streams, string input streams, and `cin`.

The second parameter to `operator<<` and `operator>>` is a reference to the `SpreadsheetCell` object that you want to write or read. The insertion operator doesn't change the `SpreadsheetCell` it writes, so that reference can be `const`. The extraction operator, however, modifies the `SpreadsheetCell` object, requiring the argument to be a non-`const` reference.

Both operators return a reference to the stream they were given as their first argument so that calls to the operator can be nested. Remember that the operator syntax is shorthand for calling the global `operator>>` or `operator<<` functions explicitly. Consider this line:

```
cin >> myCell >> anotherCell >> aThirdCell;
```

It's actually shorthand for this line:

```
operator>>(operator>>(operator>>(cin, myCell), anotherCell), aThirdCell);
```

As you can see, the return value of the first call to `operator>>` is used as input to the next. Thus, you must return the stream reference so that it can be used in the next nested call. Otherwise, the nesting won't compile.

Here are the implementations for `operator<<` and `operator>>` for the `SpreadsheetCell` class:

```
ostream& operator<<(ostream& ostr, const SpreadsheetCell& cell)
{
    ostr << cell.mString;
    return ostr;
}
istream& operator>>(istream& istr, SpreadsheetCell& cell)
{
    string temp;
    istr >> temp;
    cell.set(temp);
    return istr;
}
```

The trickiest part of these functions is that, in order for `mValue` to be set correctly, `operator>>` must remember to call the `set()` method on the `SpreadsheetCell` instead of setting `mString` directly.

OVERLOADING THE SUBSCRIPTING OPERATOR

Pretend for a few minutes that you have never heard of the `vector` or `array` class templates in the STL, and so you have decided to write your own dynamically allocated array class. This class would allow you to set and retrieve elements at specified indices, and would take care of all memory allocation “behind the scenes.” A first stab at the class definition for a dynamically allocated array might look as follows:

```

template <typename T>
class Array
{
public:
    // Creates an array with a default size that will grow as needed.
    Array();
    virtual ~Array();

    // Disallow assignment and pass-by-value
    Array<T>& operator=(const Array<T>& rhs) = delete;
    Array(const Array<T>& src) = delete;

    // Returns the value at index x. If index x does not exist in the array,
    // throws an exception of type out_of_range.
    T getElementAt(size_t x) const;

    // Sets the value at index x to val. If index x is out of range,
    // allocates more space to make it in range.
    void setElementAt(size_t x, const T& val);

private:
    static const size_t kAllocSize = 4;
    void resize(size_t newSize);
    // Sets all elements to 0
    void initializeElements();
    T* mElems;
    size_t mSize;
};

```

The interface supports setting and accessing elements. It provides random-access guarantees: a client could create an array and set elements 1, 100 and 1000 without worrying about memory management.

Here are the implementations of the methods:

```

template <typename T> Array<T>::Array()
{
    mSize = kAllocSize;
    mElems = new T[mSize];
    initializeElements();
}

template <typename T> Array<T>::~Array()
{
    delete [] mElems;
    mElems = nullptr;
}

template <typename T> void Array<T>::initializeElements()
{
    for (size_t i = 0; i < mSize; i++)
        mElems[i] = T();
}

template <typename T> void Array<T>::resize(size_t newSize)
{
    // Make a copy of the current elements pointer and size
}

```

```

        T* oldElems = mElems;
        size_t oldSize = mSize;
        // Create new bigger array
        mSize = newSize;           // store the new size
        mElems = new T[newSize];  // Allocate the new array of the new size
        initializeElements();     // Initialize all elements to 0
        // The new size is always bigger than the old size
        for (size_t i = 0; i < oldSize; i++) {
            // Copy the elements from the old array to the new one
            mElems[i] = oldElems[i];
        }
        delete [] oldElems; // free the memory for the old array
    }

    template <typename T> T Array<T>::getElementAt(size_t x) const
    {
        if (x >= mSize) {
            throw std::out_of_range("");
        }
        return mElems[x];
    }

    template <typename T> void Array<T>::setElementAt(size_t x, const T& val)
    {
        if (x >= mSize) {
            // Allocate kAllocSize past the element the client wants
            resize(x + kAllocSize);
        }
        mElems[x] = val;
    }
}

```

Here is a small example of how you could use this class:

```

Array<int> myArray;
for (size_t i = 0; i < 10; i++) {
    myArray.setElementAt(i, 100);
}
for (size_t i = 0; i < 10; i++) {
    cout << myArray.getElementAt(i) << " ";
}

```

As you can see, you never have to tell the array how much space you need. It allocates as much space as it requires to store the elements you give it. However, it's inconvenient to always use the `setElementAt()` and `getElementAt()` methods. It would be nice to be able to use conventional array index notation like this:

```

Array<int> myArray;
for (size_t i = 0; i < 10; i++) {
    myArray[i] = 100;
}
for (size_t i = 0; i < 10; i++) {
    cout << myArray[i] << " ";
}

```

This is where the overloaded subscripting operator comes in. You can add an `operator[]` to the class with the following implementation:

```
template <typename T> T& Array<T>::operator[](size_t x)
{
    if (x >= mSize) {
        // Allocate kAllocSize past the element the client wants.
        resize(x + kAllocSize);
    }
    return mElems[x];
}
```

The example code using array index notation now compiles. The `operator[]` can be used to both set and get elements because it returns a reference to the element at location `x`. This reference can be used to assign to that element. When `operator[]` is used on the left-hand side of an assignment statement, the assignment actually changes the value at location `x` in the `mElems` array.

Providing Read-Only Access with `operator[]`

Although it's sometimes convenient for `operator[]` to return an element that can serve as an lvalue, you don't always want that behavior. It would be nice to be able to provide read-only access to the elements of the array as well, by returning a `const` value or `const` reference. Ideally, you would provide two `operator[]`s: one returns a reference and one returns a `const` reference. You might try to do this as follows:

```
T& operator[](size_t x);
const T& operator[](size_t x); // Error! Can't overload based on return type.
```

However, there is one small problem: you can't overload a method or operator based only on the return type. Thus, the preceding code doesn't compile. C++ provides a way around this restriction: if you mark the second `operator[]` with the attribute `const`, then the compiler can distinguish between the two. If you call `operator[]` on a `const` object, it will use the `const` `operator[]`, and, if you call it on a non-`const` object, it will use the non-`const` `operator[]`. Here are the two operators with the correct prototypes:

```
T& operator[](size_t x);
const T& operator[](size_t x) const;
```

Here is the implementation of the `const` `operator[]`. It throws an exception if the index is out of range instead of trying to allocate new space. It doesn't make sense to allocate new space when you're only trying to read the element value:

```
template <typename T> const T& Array<T>::operator[](size_t x) const
{
    if (x >= mSize) {
        throw std::out_of_range("");
    }
    return mElems[x];
}
```

The following code demonstrates these two forms of `operator[]`:

```
void printArray(const Array<int>& arr, size_t size);
int main()
{
    Array<int> myArray;
    for (size_t i = 0; i < 10; i++) {
        myArray[i] = 100; // Calls the non-const operator[] because
                         // myArray is a non-const object.
    }
    printArray(myArray, 10);
    return 0;
}
void printArray(const Array<int>& arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        cout << arr[i] << " "; // Calls the const operator[] because arr is
                               // a const object.
    }
    cout << endl;
}
```

Note that the `const` `operator[]` is called in `printArray()` only because `arr` is `const`. If `arr` were not `const`, the non-`const` `operator[]` would be called, despite the fact that the result is not modified.

Non-Integral Array Indices

It is a natural extension of the paradigm of “indexing” into a collection by providing a key of some sort; a `vector` (or in general, any linear array) is a special case where the “key” is just a position in the array. Think of the argument of `operator[]` as providing a mapping between two domains: the domain of keys and the domain of values. Thus, you can write an `operator[]` that uses any type as its index. This type does not need to be an integer type. This is done for the STL associative containers, like `std::map`, which are described in Chapter 16.

For example, you could create an *associative array* in which you use `string` keys instead of integers. Here is the definition for an associative array class:

```
template <typename T>
class AssociativeArray
{
public:
    AssociativeArray();
    virtual ~AssociativeArray();
    T& operator[](const std::string& key);
    const T& operator[](const std::string& key) const;
private:
    // Implementation details omitted
};
```

Implementing this class would be a good exercise for you. You can also find an implementation of this class in the downloadable source code for this book at www.wrox.com/go/proc++3e.

NOTE You cannot overload the subscripting operator to take more than one parameter. If you want to provide subscripting on more than one index, you can use the function call operator explained in the next section.

OVERLOADING THE FUNCTION CALL OPERATOR

C++ allows you to overload the function call operator, written as `operator()`. If you write an `operator()` for your class, you can use objects of that class as if they were function pointers. You can overload this operator only as a non-static method in a class. Here is an example of a simple class with an overloaded `operator()` and a class method with the same behavior:

```
class FunctionObject
{
public:
    int operator() (int inParam); // function call operator
    int doSquare(int inParam); // Normal method
};

// Implementation of overloaded function call operator
int FunctionObject::operator() (int inParam)
{
    return doSquare(inParam);
}

// Implementation of normal method
int FunctionObject::doSquare(int inParam)
{
    return inParam * inParam;
}
```

Here is an example of code that uses the function call operator, contrasted with the call to a normal method of the class:

```
int x = 3, xSquared, xSquaredAgain;
FunctionObject square;
xSquared = square(x); // Call the function call operator
xSquaredAgain = square.doSquare(x); // Call the normal method
```

An object of a class with a function call operator is called a *function object*, or *functor*, for short.

At first, the function call operator probably seems a little strange. Why would you want to write a special method for a class to make objects of the class look like function pointers? Why wouldn't you just write a function or a standard method of a class? The advantage of function objects over standard methods of objects is simple: these objects can sometimes masquerade as function pointers. You can pass function objects as callback functions to routines that expect function pointers, as long as the function pointer types are templated. This is discussed in more detail in Chapter 17.

The advantages of function objects over global functions are more intricate. There are two main benefits:

- Objects can retain information in their data members between repeated calls to their function call operators. For example, a function object might be used to keep a running sum of numbers collected from each call to the function call operator.

- You can customize the behavior of a function object by setting data members. For example, you could write a function object to compare an argument to the function against a data member. This data member could be configurable so that the object could be customized for whatever comparison you want.

Of course, you could implement either of the preceding benefits with global or `static` variables. However, function objects provide a cleaner way to do it, and using global or `static` variables might cause problems in a multithreaded application. The true benefits of function objects are demonstrated with the STL in Chapter 17.

By following the normal method overloading rules, you can write as many `operator()`s for your classes as you want. Specifically, the various `operator()`s must have a different number of parameters or different types of parameters. For example, you could add an `operator()` to the `FunctionObject` class that takes a `string` reference:

```
int operator() (int iParam);
void operator() (string& str);
```

The function call operator can also be used to provide subscripting for multiple indices of an array. Simply write an `operator()` that behaves like `operator[]` but allows more than one parameter. The only problem with this technique is that now you have to use `()` to index instead of `[]`, as in `myArray(3, 4) = 6;`

OVERLOADING THE DEREFERENCING OPERATORS

You can overload three de-referencing operators: `*`, `->`, and `->*`. Ignoring `->*` for the moment (I'll come back to it later), consider the built-in meanings of `*` and `->`. `*` dereferences a pointer to give you direct access to its value, while `->` is shorthand for a `*` dereference followed by a `.` member selection. The following code shows the equivalences:

```
SpreadsheetCell* cell = new SpreadsheetCell;
(*cell).set(5); // Dereference plus member selection
cell->set(5); // Shorthand arrow dereference and member selection together
```

You can overload the dereferencing operators for your classes in order to make objects of the classes behave like pointers. The main use of this capability is for implementing smart pointers, introduced in Chapter 1. It is also useful for iterators, which the STL uses, discussed in Chapter 16. This chapter teaches you the basic mechanics for overloading the relevant operators in the context of a simple smart pointer class template.

WARNING *C++ has two standard smart pointers called `std::unique_ptr` and `std::shared_ptr`. It is highly recommended to use these standard smart pointer classes instead of writing your own. The example here is given only to demonstrate how to write dereferencing operators.*

Here is the example smart pointer class template definition, without the dereference operators filled in yet:

```
template <typename T> class Pointer
{
public:
    Pointer(T* inPtr);
    virtual ~Pointer();
    // Prevent assignment and pass by value.
    Pointer(const Pointer<T>& src) = delete;
    Pointer<T>& operator=(const Pointer<T>& rhs) = delete;

    // Dereference operators will go here.
private:
    T* mPtr;
};
```

This smart pointer is about as simple as you can get. All it does is store a dumb pointer, and the storage pointed to by the pointer is deleted when the smart pointer is destroyed. The implementation is equally simple: the constructor takes a real (“dumb”) pointer, which is stored as the only data member in the class. The destructor frees the storage referenced by the pointer:

```
template <typename T> Pointer<T>::Pointer(T* inPtr) : mPtr(inPtr)
{
}
template <typename T> Pointer<T>::~Pointer()
{
    delete mPtr;
    mPtr = nullptr;
}
```

You would like to be able to use the smart pointer template like this:

```
Pointer<int> smartInt(new int);
*smartInt = 5; // Dereference the smart pointer.
cout << *smartInt << endl;
Pointer<SpreadsheetCell> smartCell(new SpreadsheetCell);
smartCell->set(5); // Dereference and member select the set method.
cout << smartCell->getValue() << endl;
```

As you can see from this example, you will have to provide implementations of `operator*` and `operator->` for this class. These will be implemented in the next two sections.

WARNING *You should rarely, if ever, write an implementation of just one of operator* and operator->. You should almost always write both operators together. It would confuse the users of your class if you failed to provide both.*

Implementing operator*

When you dereference a pointer, you expect to be able to access the memory to which the pointer points. If that memory contains a simple type such as an `int`, you should be able to change its value

directly. If the memory contains a more complicated type, such as an object, you should be able to access its data members or methods with the `.` operator.

To provide these semantics, you should return a reference to a variable or object from `operator*`. In the `Pointer` class, the declaration and definition look like this:

```
template <typename T> class Pointer
{
public:
    // Omitted for brevity
    T& operator*();
    const T& operator*() const;
    // Omitted for brevity
};
template <typename T> T& Pointer<T>::operator*()
{
    return *mPtr;
}
template <typename T> const T& Pointer<T>::operator*() const
{
    return *mPtr;
}
```

As you can see, `operator*` returns a reference to the object or variable to which the underlying dumb pointer points. As with overloading the subscripting operators, it's useful to provide both `const` and non-`const` versions of the method, which return a `const` reference and a non-`const` reference, respectively.

Implementing `operator->`

The arrow operator is a bit trickier. The result of applying the arrow operator should be a member or method of an object. However, in order to implement it like that, you would have to be able to implement the equivalent of `operator*` followed by `operator.`; C++ doesn't allow you to overload `operator.` for good reason: it's impossible to write a single prototype that allows you to capture any possible member or method selection. Therefore, C++ treats `operator->` as a special case.

Consider this line:

```
smartCell->set(5);
```

C++ translates this to:

```
(smartCell.operator->())->set(5);
```

As you can see, C++ applies another `operator->` to whatever you return from your overloaded `operator->`. Therefore, you must return a pointer to an object, like this:

```
template <typename T> class Pointer
{
public:
    // Omitted for brevity
    T* operator->();
    const T* operator->() const;
    // Omitted for brevity
};
```

```
template <typename T> T* Pointer<T>::operator->()
{
    return mPtr;
}
template <typename T> const T* Pointer<T>::operator->() const
{
    return mPtr;
}
```

You may find it confusing that `operator*` and `operator->` are asymmetric, but, once you see them a few times, you'll get used to it.

What in the World Is `operator->*` ?

It's perfectly legitimate in C++ to take the addresses of class data members and methods in order to obtain pointers to them. However, you can't access a non-static data member or call a non-static method without an object. The whole point of class data members and methods is that they exist on a per-object basis. Thus, when you want to call the method or access the data member via the pointer, you must dereference the pointer in the context of an object. The following example demonstrates this. Chapter 22 discusses the syntactical details in the section called "Pointers to Methods and Members." You can ignore these details for this example; the only important parts for now are the `.*` and the `->*` operators:

```
SpreadsheetCell myCell;
double (SpreadsheetCell::*methodPtr) () const = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr) () << endl;
```

Note the use of the `.*` operator to dereference the method pointer and call the method. There is also an equivalent `operator->*` for calling methods via pointers when you have a pointer to an object instead of the object itself. The operator looks like this:

```
SpreadsheetCell* myCell = new SpreadsheetCell();
double (SpreadsheetCell::*methodPtr) () const = &SpreadsheetCell::getValue;
cout << (myCell->*methodPtr) () << endl;
```

C++ does not allow you to overload `operator.*` (just as you can't overload `operator.`), but you could overload `operator->*`. However, it is very tricky, and, given that most C++ programmers don't even know that you can access methods and data members through pointers, it's probably not worth the trouble. The `shared_ptr` template in the standard library, for example, does not overload `operator->*`.

WRITING CONVERSION OPERATORS

Going back to the `SpreadsheetCell` example, consider these two lines of code:

```
SpreadsheetCell cell(1.23);
string str = cell; // DOES NOT COMPILE!
```

A `SpreadsheetCell` contains a string representation, so it seems logical that you could assign it to a `string` variable. Well, you can't. The compiler tells you that it doesn't know how to convert a `SpreadsheetCell` to a `string`. You might be tempted to try forcing the compiler to do what you want, like this:

```
string str = (string)cell; // STILL DOES NOT COMPILE!
```

First, the preceding code still doesn't compile because the compiler still doesn't know *how* to convert the `SpreadsheetCell` to a string. It already knew from the first line what you wanted it to do, and it would do it if it could. Second, it's a bad idea in general to add gratuitous casts to your program. If you want to allow this kind of assignment, you must tell the compiler how to perform it. Specifically, you can write a conversion operator to convert `SpreadsheetCells` to strings. The prototype looks like this:

```
operator std::string() const;
```

The name of the function is `operator std::string`. It has no return type because the return type is specified by the name of the operator: `std::string`. It is `const` because it doesn't change the object on which it is called. The implementation looks as follows:

```
SpreadsheetCell::operator string() const
{
    return mString;
}
```

That's all you need to do to write a conversion operator from `SpreadsheetCell` to `string`. Now the compiler accepts the following lines and does the right thing at run time:

```
SpreadsheetCell cell(1.23);
string str = cell; // Works as expected
```

You can write conversion operators for any type with this same syntax. For example, here is a `double` conversion operator for `SpreadsheetCell`:

```
SpreadsheetCell::operator double() const
{
    return mValue;
}
```

Now you can write code like the following:

```
SpreadsheetCell cell(1.23);
double d1 = cell;
```

Ambiguity Problems with Conversion Operators

Note that writing the `double` conversion operator for the `SpreadsheetCell` object introduces an *ambiguity* problem. Consider this line:

```
SpreadsheetCell cell(1.23);
double d2 = cell + 3.3; // DOES NOT COMPILE IF YOU DEFINE operator double()
```

This line now fails to compile. It worked before you wrote `operator double()`, so what's the problem now? The issue is that the compiler doesn't know if it should convert `cell` to a `double` with `operator double()` and perform `double` addition, or convert `3.3` to a `SpreadsheetCell` with the `double` constructor and perform `SpreadsheetCell` addition. Before you wrote `operator double()`, the compiler had only one choice: Convert `3.3` to a `SpreadsheetCell` with the `double` constructor and perform `SpreadsheetCell` addition. However, now the compiler could do either. It doesn't want to make a choice you might not like, so it refuses to make any choice at all.

The usual pre-C++11 solution to this conundrum is to make the constructor in question `explicit`, so that the automatic conversion using that constructor is prevented. However, we don't want that constructor to be `explicit` because we generally like the automatic conversion of `doubles` to `SpreadsheetCells`, as explained in Chapter 8. Since C++11, you can solve this problem by making the `double` conversion operator `explicit`:

```
explicit operator double() const;
```

The following code demonstrates its use:

```
SpreadsheetCell cell = 6.6;           // [1]
string str = cell;                   // [2]
double d1 = static_cast<double>(cell); // [3]
double d2 = static_cast<double>(cell + 3.3); // [4]
```

- [1] Uses the implicit conversion from a `double` to a `SpreadsheetCell`. Because this is in the declaration, this is done by calling the constructor that accepts a `double`.
- [2] Uses the `operator string()` conversion operator.
- [3] Uses the `operator double()` conversion operator. Note that because this conversion operator is now declared `explicit`, the cast is required.
- [4] Uses the implicit conversion of `3.3` to a `SpreadsheetCell`, followed by `operator+` on two `SpreadsheetCells`, followed by a required explicit cast to invoke `operator double()`.

Conversions for Boolean Expressions

Sometimes it is useful to be able to use objects in Boolean expressions. For example, programmers often use pointers in conditional statements like this:

```
if (ptr != nullptr) { /* Perform some dereferencing action. */ }
```

Sometimes they write shorthand conditions such as:

```
if (ptr) { /* Perform some dereferencing action. */ }
```

Other times, you see code as follows:

```
if (!ptr) { /* Do something. */ }
```

Currently, none of the preceding expressions compile with the `Pointer` smart pointer class defined earlier. However, you can add a conversion operator to the class to convert it to a pointer type. Then, the comparisons to `nullptr`, as well as the object alone in an `if` statement, will trigger the conversion to the pointer type. The usual pointer type for the conversion operator is `void*` because that is a pointer type with which you cannot do much except testing it in Boolean expressions.

```
operator void*() const { return mPtr; }
```

Now the following code compiles and does what you expect:

```
void process(Pointer<SpreadsheetCell>& p)
{
    if (p != nullptr) { cout << "not nullptr" << endl; }
    if (p != NULL) { cout << "not NULL" << endl; }
    if (p) { cout << "not nullptr" << endl; }
    if (!p) { cout << "nullptr" << endl; }
```

```

    }
int main()
{
    Pointer<SpreadsheetCell> smartCell(nullptr);
    process(smartCell);
    cout << endl;
    Pointer<SpreadsheetCell> anotherSmartCell(new SpreadsheetCell(5.0));
    process(anotherSmartCell);
}

```

The output is as follows:

```

nullptr

not nullptr
not NULL
not nullptr

```

Another alternative is to overload `operator bool()` as follows instead of `operator void*()`. After all, you're using the object in a Boolean expression; why not convert it directly to a `bool`?

```
operator bool() const { return mPtr != nullptr; }
```

The following comparisons still work:

```

if (p != NULL) { cout << "not NULL" << endl; }
if (p) { cout << "not nullptr" << endl; }
if (!p) { cout << "nullptr" << endl; }

```

However, with `operator bool()`, the following comparison with `nullptr` results in a compiler error:

```
if (p != nullptr) { cout << "not nullptr" << endl; } // Error
```

This is correct behavior because `nullptr` has its own type called `nullptr_t`, which is not automatically converted to the integer 0. The compiler cannot find an `operator!=` that takes a `Pointer` object and a `nullptr_t` object. You could implement such an `operator!=` as a `friend` of the `Pointer` class:

```

template <typename T>
bool operator!=(const Pointer<T>& lhs, const std::nullptr_t& rhs)
{
    return lhs.mPtr != rhs;
}

```

However, after implementing this `operator!=`, the following comparison stops working, because the compiler doesn't know anymore which `operator!=` to use.

```
if (p != NULL) { cout << "not NULL" << endl; }
```

From this example, you might conclude that the `operator bool()` technique seems only appropriate for objects that don't represent pointers and for which conversion to a pointer type really doesn't make sense. Unfortunately, adding a conversion operator to `bool` presents some other unanticipated consequences. C++ applies "promotion" rules to silently convert `bool` to `int` whenever the opportunity arises. Therefore, with the `operator bool()`, the following code compiles and runs:

```

Pointer<SpreadsheetCell> smartCell(new SpreadsheetCell);
int i = smartCell; // Converts smartCell Pointer to bool to int.

```

That's usually not behavior that you expect or desire. Thus, many programmers prefer `operator void*()` instead of `operator bool()`.

As you can see, there is a design element to overloading operators. Your decisions about which operators to overload directly influence the ways in which clients can use your classes.

OVERLOADING THE MEMORY ALLOCATION AND DEALLOCATION OPERATORS

C++ gives you the ability to redefine the way memory allocation and deallocation work in your programs. You can provide this customization both on the global level and the class level. This capability is most useful when you are worried about memory fragmentation, which can occur if you allocate and deallocate a lot of small objects. For example, instead of going to the default C++ memory allocation each time you need memory, you could write a memory pool allocator that reuses fixed-size chunks of memory. This section explains the subtleties of the memory allocation and deallocation routines and shows you how to customize them. With these tools, you should be able to write your own allocator if the need ever arises.

WARNING *Unless you know a lot about memory allocation strategies, attempts to overload the memory allocation routines are rarely worth the trouble. Don't overload them just because it sounds like a neat idea. Only do so if you have a genuine requirement and the necessary knowledge.*

How new and delete Really Work

One of the trickiest aspects of C++ is the details of `new` and `delete`. Consider this line of code:

```
SpreadsheetCell* cell = new SpreadsheetCell();
```

The part “`new SpreadsheetCell()`” is called the *new-expression*. It does two things. First, it allocates space for the `SpreadsheetCell` object by making a call to operator `new`. Second, it calls the constructor for the object. Only after the constructor has completed does it return the pointer to you.

`delete` works analogously. Consider this line of code:

```
delete cell;
```

This line is called the *delete-expression*. It first calls the destructor for `cell`, then calls operator `delete` to free the memory.

You can overload operator `new` and operator `delete` to control memory allocation and deallocation, but you cannot overload the *new-expression* or the *delete-expression*. Thus, you can customize the actual memory allocation and deallocation, but not the calls to the constructor and destructor.

The New-Expression and operator new

There are six different forms of the *new-expression*, each of which has a corresponding operator `new`. Earlier chapters in this book already show four new-expressions: `new`, `new[]`, `nothrow new`, and `nothrow new[]`. The following list shows the corresponding four `operator new` forms from the `<new>` header file:

```
void* operator new(size_t size);                                // For new
void* operator new[](size_t size);                             // For new[]
void* operator new(size_t size, const noexcept_t&) noexcept; // For noexcept new
void* operator new[](size_t size, const noexcept_t&) noexcept; // For noexcept new[]
```

There are two special new-expressions that do no allocation, but invoke the constructor on an existing piece of storage. These are called *placement new operators* (including both single and array forms). They allow you to construct an object in preexisting memory like this:

```
void* ptr = allocateMemorySomehow();
SpreadsheetCell* cell = new (ptr) SpreadsheetCell();
```

This feature is a bit obscure, but it's important to realize that it exists. It can come in handy if you want to implement memory pools such that you reuse memory without freeing it in between. The corresponding operator `new` forms look as follows, however, the C++ standard forbids you from overloading these.

```
void* operator new(size_t size, void* p) noexcept;
void* operator new[](size_t size, void* p) noexcept;
```

The Delete-Expression and operator delete

There are only two different forms of the *delete-expression* that you can call: `delete`, and `delete[]`; there are no `nothrow` or placement forms. However, there are all six forms of operator `delete`. Why the asymmetry? The two `nothrow` and two placement forms are used only if an exception is thrown from a constructor. In that case, the operator `delete` is called that matches the operator `new` that was used to allocate the memory prior to the constructor call. However, if you delete a pointer normally, `delete` will call either operator `delete` or operator `delete[]` (never the `nothrow` or placement forms). Practically, this doesn't really matter: the C++ standard says that throwing an exception from `delete` results in undefined behavior, which means `delete` should never throw an exception anyway, so the `nothrow` version of operator `delete` is superfluous; and placement `delete` should be a no-op, because the memory wasn't allocated in placement operator `new`, so there's nothing to free. Here are the prototypes for the operator `delete` forms:

```
void operator delete(void* ptr) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete(void* ptr, const noexcept_t&) noexcept;
void operator delete[](void* ptr, const noexcept_t&) noexcept;
void operator delete(void* p, void*) noexcept;
void operator delete[](void* p, void*) noexcept;
```

Overloading operator new and operator delete

You can actually replace the global operator `new` and operator `delete` routines if you want. These functions are called for every new-expression and delete-expression in the program, unless there are more specific routines in individual classes. However, to quote Bjarne Stroustrup, “...

replacing the global operator `new` and operator `delete` is not for the fainthearted.” (*The C++ Programming Language*, third edition, Addison-Wesley, 1997). I don’t recommend it either!

WARNING *If you fail to heed my advice and decide to replace the global operator `new`, keep in mind that you cannot put any code in the operator that makes a call to `new` because this will cause an infinite loop. For example, you cannot write a message to the console with `cout`.*

A more useful technique is to overload operator `new` and operator `delete` for specific classes. These overloaded operators will be called only when you allocate and deallocate objects of that particular class. Here is an example of a class that overloads the four non-placement forms of operator `new` and operator `delete`:

```
#include <new>
class MemoryDemo
{
public:
    MemoryDemo() {}
    virtual ~MemoryDemo() {}
    void* operator new(std::size_t size);
    void operator delete(void* ptr) noexcept;
    void* operator new[] (std::size_t size);
    void operator delete[] (void* ptr) noexcept;
    void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
    void operator delete(void* ptr, const std::nothrow_t&) noexcept;
    void* operator new[] (std::size_t size, const std::nothrow_t&) noexcept;
    void operator delete[] (void* ptr, const std::nothrow_t&) noexcept;
};
```

Here are simple implementations of these operators that pass the arguments through to calls to the global versions of the operators. Note that `nothrow` is actually a variable of type `nothrow_t`:

```
void* MemoryDemo::operator new(size_t size)
{
    cout << "operator new" << endl;
    return ::operator new(size);
}
void MemoryDemo::operator delete(void* ptr) noexcept
{
    cout << "operator delete" << endl;
    ::operator delete(ptr);
}
void* MemoryDemo::operator new[] (size_t size)
{
    cout << "operator new[]" << endl;
    return ::operator new[] (size);
}
void MemoryDemo::operator delete[] (void* ptr) noexcept
{
    cout << "operator delete[]" << endl;
    ::operator delete[] (ptr);
```

```

    }
    void* MemoryDemo::operator new(size_t size, const noexcept_t&) noexcept
    {
        cout << "operator new nothrow" << endl;
        return ::operator new(size, noexcept);
    }
    void MemoryDemo::operator delete(void* ptr, const noexcept_t&) noexcept
    {
        cout << "operator delete nothrow" << endl;
        ::operator delete(ptr, noexcept);
    }
    void* MemoryDemo::operator new[](size_t size, const noexcept_t&) noexcept
    {
        cout << "operator new[] nothrow" << endl;
        return ::operator new[](size, noexcept);
    }
    void MemoryDemo::operator delete[](void* ptr, const noexcept_t&) noexcept
    {
        cout << "operator delete[] nothrow" << endl;
        ::operator delete[](ptr, noexcept);
    }
}

```

Here is some code that allocates and frees objects of this class in several ways:

```

MemoryDemo* mem = new MemoryDemo();
delete mem;
mem = new MemoryDemo[10];
delete [] mem;
mem = new (nothrow) MemoryDemo();
delete mem;
mem = new (nothrow) MemoryDemo[10];
delete [] mem;

```

Here is the output from running the program:

```

operator new
operator delete
operator new[]
operator delete[]
operator new nothrow
operator delete
operator new[] nothrow
operator delete[]

```

These implementations of `operator new` and `operator delete` are obviously trivial and not particularly useful. They are intended only to give you an idea of the syntax in case you ever want to implement nontrivial versions of them.

WARNING *Whenever you overload operator new, overload the corresponding form of operator delete. Otherwise, memory will be allocated as you specify but freed according to the built-in semantics, which may not be compatible.*

It might seem overkill to overload all of the various forms of operator `new`. However, it's generally a good idea to do so in order to prevent inconsistencies in the memory allocations. If you don't want to provide implementations, you can explicitly delete the function using `=delete` in order to prevent anyone from using it. See the next section.

WARNING *Overload all forms of operator new, or explicitly delete forms that you don't want to get used.*

Explicitly Deleting/Defaulting operator new and operator delete

Chapter 7 shows how you can explicitly delete or default a constructor or assignment operator. Explicitly deleting or defaulting is not limited to constructors and assignment operators. For example, the following class deletes the operator `new` and `new[]`, which means that this class cannot be dynamically created by using `new` or `new[]`:

```
class MyClass
{
public:
    void* operator new(std::size_t size) = delete;
    void* operator new[] (std::size_t size) = delete;
};
```

Using this class in the following ways will result in compiler errors:

```
int main()
{
    MyClass* p1 = new MyClass;
    MyClass* pArray = new MyClass[2];
    return 0;
}
```

Overloading operator new and operator delete with Extra Parameters

In addition to overloading the standard forms of operator `new`, you can write your own versions with extra parameters. For example, here are the prototypes for an additional operator `new` and operator `delete` with an extra integer parameter for the `MemoryDemo` class:

```
void* operator new(std::size_t size, int extra);
void operator delete(void* ptr, int extra) noexcept;
```

The implementation is as follows:

```
void* MemoryDemo::operator new(std::size_t size, int extra)
{
    cout << "operator new with extra int arg: " << extra << endl;
    return ::operator new(size);
}
void MemoryDemo::operator delete(void* ptr, int extra) noexcept
```

```

{
    cout << "operator delete with extra int arg: " << extra << endl;
    return ::operator delete(ptr);
}

```

When you write an overloaded `operator new` with extra parameters, the compiler will automatically allow the corresponding `new`-expression. So, you can now write code like this:

```

MemoryDemo* memp = new(5) MemoryDemo();
delete memp;

```

The extra arguments to `new` are passed with function call syntax (as in `nothrow new`). These extra arguments can be useful for passing various flags or counters to your memory allocation routines. For example, some runtime libraries use this in debug mode to provide the file name and line number where an object is allocated, so when there is a memory leak, the offending line that did the allocation can be identified.

When you define an `operator new` with extra parameters, you should also define the corresponding `operator delete` with the same extra parameters. You cannot call this `operator delete` with extra parameters yourself, but it will be called only when you use your `operator new` with extra parameters and the constructor of your object throws an exception.

An alternate form of `operator delete` gives you the size of the memory that should be freed as well as the pointer. Simply declare the prototype for `operator delete` with an extra size parameter.

WARNING *If your class declares two identical versions of `operator delete` except that one takes the size parameter and the other doesn't, the version without the size parameter will always get called. If you want the version with the size parameter to be used, write only that version.*

You can replace `operator delete` with the version that takes a size for any of the versions of `operator delete` independently. Here is the `MemoryDemo` class definition with the first `operator delete` modified to take the size of the memory to be deleted:

```

class MemoryDemo
{
public:
    // Omitted for brevity
    void* operator new(std::size_t size);
    void operator delete(void* ptr, std::size_t size) noexcept;
    // Omitted for brevity
};

```

The implementation of this `operator delete` calls the global `operator delete` without the size parameter because there is no global `operator delete` that takes the size:

```

void MemoryDemo::operator delete(void* ptr, std::size_t size) noexcept
{
    cout << "operator delete with size" << endl;
    ::operator delete(ptr);
}

```

This capability is useful only if you are writing a complicated memory allocation and deallocation scheme for your classes.

SUMMARY

This chapter summarized the rationale for operator overloading and provided examples and explanations for overloading the various categories of operators. Hopefully, this chapter taught you to appreciate the power that it gives you. Throughout this book, operator overloading is used to provide abstractions and easy-to-use class interfaces.

Now it's time to start delving into the C++ Standard Library. The next chapter starts with an overview of the functionality provided by the C++ Standard Library, followed by chapters that go deeper in on specific features of the library.

15

Overview of the C++ Standard Library

WHAT'S IN THIS CHAPTER?

- The coding principles used throughout the standard library
- The kind of functionality the standard library provides

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

The most important library that you will use as a C++ programmer is the C++ standard library. As its name implies, this library is part of the C++ standard, so any standards-conforming compiler should include it. The standard library is not monolithic: It includes several disparate components, some of which you have been using already. You may even have assumed they were part of the core language. All standard library classes and functions are declared in the `std` namespace.

The heart of the C++ standard library is its generic *containers* and *algorithms*. This subset of the library is often called the *Standard Template Library*, or STL for short, because of its abundant use of templates. The power of the STL is that it provides generic containers and generic algorithms in such a way that most of the algorithms work on most of the containers, no matter what type of data the containers store. Performance is a very important part of the STL. The goal is to make the STL containers and algorithms as fast as or faster than hand-written code.

The C++ standard library also includes all C headers that are part of the C99 standard, but with new names. For example, you can access the functionality from the C `<stdio.h>`

header by including `<cstdio>`. However, the use of functionality provided by these C headers is discouraged in favor of true C++ functionality.

A C++ programmer who wishes to claim language expertise is expected to be familiar with the standard library. You can save yourself immeasurable time and energy by incorporating STL containers and algorithms into your programs instead of writing and debugging your own versions. Now is the time to master the standard library.

This first chapter on the standard library provides a general overview of the functionality available in the standard library and in the STL.

The next few chapters go into more detail on several aspects of the standard library and the STL, including containers, iterators, generic algorithms, predefined function object classes, regular expressions, additional library utilities, and customizing and extending the library.

Despite the depth of material found in this and the next chapters, the standard library is too large for this book to cover exhaustively. You should read this and the following chapters to learn about the standard library and the STL, but keep in mind that they don't mention every method and data member that the various classes provide, or show you the prototypes of every algorithm. Appendix C provides a summary of all the header files in the standard library. Consult a Standard Library Reference, for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>, for a complete reference of all the provided functionality.

CODING PRINCIPLES

The standard library, and definitely the standard template library, make heavy use of the C++ features called *templates* and *operator overloading*.

Use of Templates

Templates are used to allow *generic programming*. They make it possible to write code that can work with all kinds of objects, even objects unknown to the programmer when writing the code. The obligation of the programmer writing the template code is to specify the requirements of the classes that define these objects; for example, that they have an operator for comparison, or a copy constructor, or whatever is deemed appropriate, and then making sure the code that is written uses only those required capabilities. The obligation of the programmer creating the objects is to supply those operators and methods that the template writer requires.

Unfortunately, many programmers consider templates to be the most difficult part of C++ and, for that reason, tend to avoid them. However, even if you never write your own templates, you need to understand their syntax and capabilities in order to use the STL. Templates are described in detail in Chapter 11. If you skipped that chapter and are not familiar with templates, I suggest you first read Chapter 11 and then come back to learn about the standard library.

Use of Operator Overloading

Operator overloading is another feature used extensively by the C++ standard library, including the STL. Chapter 8 has a whole section devoted to operator overloading. Make sure you read that

section and understand it before tackling this and subsequent chapters. In addition, Chapter 14 presents much more detail on the subject of operator overloading, but those details are not required to understand the following chapters.

OVERVIEW OF THE C++ STANDARD LIBRARY

This section introduces the various components of the standard library from a design perspective. You will learn what facilities are available for you to use, but you will not learn the coding details. Those details are covered in other chapters.

Strings

C++ provides a built-in `string` class, defined in the `<string>` header. Although you may still use C-style strings of character arrays, the C++ `string` class is superior in almost every way. It handles the memory management; provides some bounds checking, assignment semantics, and comparisons; and supports manipulations such as concatenation, substring extraction, and substring or character replacement.

NOTE *Technically, the C++ `string` is a `typedef` name for a `char` instantiation of the `basic_string` template. However, you need not worry about these details; you can use `string` as if it were a bona fide non-template class.*

C++ provides support for *Unicode* and *localization*. These features allow you to write programs that work with different languages, for example Chinese. Locales, defined in `<locale>`, allow you to format data such as numbers and dates according to the rules of a certain country or region.

In case you missed it, Chapter 2 provides all the details of the `string` class functionality, while Chapter 18 discusses Unicode and localization.

Regular Expressions

Regular expressions are available through the `<regex>` header file. They make it easy to perform so-called *pattern-matching*, often used in text processing. Pattern-matching allows you to search special patterns in strings and optionally replace those with a new pattern. Regular expressions are discussed in Chapter 18.

I/O Streams

C++ introduces a new model for input and output using *streams*. The C++ library provides routines for reading and writing built-in types from and to files, console/keyboard, and strings. C++ also provides the facilities for coding your own routines for reading and writing your own objects. The I/O functionality is defined in several header files: `<fstream>`, `<iomanip>`, `<ios>`, `<iosfwd>`, `<iostream>`, `<iostream>`, `<ostream>`, `<sstream>`, `<streambuf>`, and `<strstream>`. Chapter 1 reviews the basics of I/O streams, and Chapter 12 discusses streams in detail.

Smart Pointers

One of the problems faced in robust programming is knowing when to delete an object. There are several failures that can happen. A first problem is not deleting the object at all (failing to free the storage). This is known as *memory leaks*, where objects accumulate and take up space but are not used. Another problem is where someone deletes the storage but others are still pointing to that storage, resulting in pointers to storage that is either no longer in use or has been reallocated for another purpose. These are known as *dangling pointers*. One more problem is when one piece of code frees the storage, and another piece of code attempts to free the same storage. This is known as *double-freeing*. All of these tend to result in program failures of some sort. Some failures are readily detected; others cause the program to produce erroneous results. Most of these errors are difficult to discover and repair.

C++ addresses these problems with smart pointers: `unique_ptr`, `shared_ptr`, and `weak_ptr`. `shared_ptr` and `weak_ptr` are thread-safe. They are all defined in the `<memory>` header. These smart pointers are introduced in Chapter 1 and discussed in more detail in Chapter 22.

Before C++11, the functionality of `unique_ptr` was handled by a type called `auto_ptr`, which is now deprecated and should not be used anymore. There was no equivalent to `shared_ptr` in the earlier standard library, although many third-party libraries (for example, Boost) did provide this capability.

Exceptions

The C++ language supports exceptions, which allow functions or methods to pass errors of various types up to calling functions or methods. The C++ standard library provides a class hierarchy of exceptions that you can use in your program as is, or that you can derive from to create your own exception types. Exception support is defined in a couple of header files: `<exception>`, `<stdexcept>`, and `<system_error>`. Chapter 13 covers the details of exceptions and the standard exception classes.

Mathematical Utilities

The C++ library provides some mathematical utility classes. Although they are templated so that you can use them with any type, they are not generally considered part of the STL.

The standard library provides a complex number class called `complex`, defined in `<complex>`, which provides an abstraction for working with numbers that contain both real and imaginary components.

The compile-time rational arithmetic library provides a `ratio` class template defined in the `<ratio>` header file. This `ratio` class can exactly represent any finite rational number defined by a numerator and denominator. This library is discussed in Chapter 19.

The standard library also contains a class called `valarray`, defined in `<valarray>`, which is similar to the `vector` class but is more optimized for high performance numerical applications. The library provides several related classes to represent the concept of vector slices. From these building blocks, it is possible to build classes to perform matrix mathematics. There is no built-in matrix class; however, there are third-party libraries like Boost that include matrix support.

C++ also provides a standard way to obtain information about numeric limits, such as the maximum possible value for an integer on the current platform. In C, you could access `#defines`, such as `INT_MAX`. While those are still available in C++, it's recommended to use the `numeric_limits` class template defined in the `<limits>` header file. Its use is straightforward, as shown in the following code:

```
cout << "Max int value: " << numeric_limits<int>::max() << endl;
cout << "Min int value: " << numeric_limits<int>::min() << endl;
cout << "Lowest int value: " << numeric_limits<int>::lowest() << endl;

cout << "Max double value: " << numeric_limits<double>::max() << endl;
cout << "Min double value: " << numeric_limits<double>::min() << endl;
cout << "Lowest double value: " << numeric_limits<double>::lowest() << endl;
```

Time Utilities

C++ includes the Chrono library, defined in the `<chrono>` header file. This library makes it easy to work with time; for example, to time certain durations or to perform actions based on timing. The Chrono library is discussed in detail in Chapter 19.

Random Numbers

C++ already had support for generating pseudo-random numbers for a long time with the `rand()` and `rand()` functions. However, those functions provide only very basic random numbers. For example, you could not change the distribution of the generated random numbers.

Since C++11, a random number library has been added to the standard, which is much more powerful. The new library is defined in `<random>` and comes with *random number engines*, *random number engine adaptors*, and *random number distributions*. All of these can be used to give you random numbers more suited to your problem domain, such as normal distributions, negative exponential distributions, etc.

Consult Chapter 19 for details on this library.

Initializer Lists

Initializer lists are defined in the `<initializer_list>` header file. They make it easy to write functions that can accept a variable number of arguments and are discussed in Chapter 10.

Pair and Tuple

The `<utility>` header defines the `pair` template, which can store two elements with two different types. This is known as storing *heterogeneous* elements. All standard template library containers discussed further in this chapter store *homogenous* elements, meaning that all the elements in a container must have the same type. A `pair` allows you to store elements of completely unrelated types in one object.

`tuple` defined in `<tuple>` is a generalization of `pair`. It is a sequence with a fixed size that can have heterogeneous elements. The number and type of elements for a `tuple` instantiation is fixed at compile time. Tuples are discussed in Chapter 19.

Function Objects

A class that implements a function call operator is called a *function object*. Function objects can, for example, be used as predicates for certain STL algorithms. The `<functional>` header file defines a number of predefined function objects and supports creating new function objects based on existing ones.

Function objects are discussed in detail in Chapter 17 together with a detailed discussion of the STL algorithms.

Multithreading

All major CPU vendors are selling processors with multiple cores. They are being used for everything from servers to consumer computers and even in smartphones. If you want your software to take advantage of all these cores then you need to write multithreaded code. The standard library provides a couple of basic building blocks for writing such code. Individual threads can be created using the `thread` class from the `<thread>` header. In multithreaded code you need to take care that several threads are not reading and writing to the same piece of data at the same time. To prevent this, you can use atomics defined in `<atomic>`, which give you thread-safe atomic access to a piece of data. Other thread synchronization mechanisms are provided by `<condition_variable>` and `<mutex>`.

If you just need to calculate something and get the result back with proper exception handling, it's easier to use `async` and `future` defined in the `<future>` header than it is to directly use the `thread` class.

Writing multithreaded code is discussed in detail in Chapter 23.

Type Traits

Type traits are defined in the `<type_traits>` header file and provide information about types at compile time. They are useful when writing advanced templates and are discussed in Chapter 21.

The Standard Template Library

The standard template library (STL) supports various *containers* and *algorithms*. This section briefly introduces those containers and algorithms. Later chapters provide the coding details for using them in your programs.

STL Containers

The STL provides implementations of commonly used data structures such as linked lists and queues. When you use C++, you should not need to write such data structures again. The data structures are implemented using a concept called *containers*, which store information called *elements*, in a way that implements the data structure (linked list, queue, etc.) appropriately. Different data structures have different insertion, deletion, and access behavior and performance characteristics. It is important to be familiar with the data structures available so that you can choose the most appropriate one for any given task.

All the containers in the STL are templates, so you can use them to store any type, from built-in types such as `int` and `double` to your own classes. Each container instance stores objects of only one type; that is, they are *homogeneous collections*. If you need non fixed-sized heterogeneous collections, you could create a class that has multiple derived classes, and each derived class could wrap an object of your required type.

NOTE *The C++ STL containers are homogenous: they allow elements of only one type in each container.*

Note that the C++ standard specifies the *interface*, but not the *implementation*, of each container and algorithm. Thus, different vendors are free to provide different implementations. However, the standard also specifies performance requirements as part of the interface, which the implementations must meet.

This section provides an overview of the various containers available in the STL.

vector

The `<vector>` header file defines `vector`, which stores a sequence of elements and provides random access to these elements. You can think of a `vector` as an array of elements that grows dynamically as you insert elements, and provides some bounds checking. Like an array, the elements of a `vector` are stored in contiguous memory.

NOTE *A vector in C++ is a synonym for a dynamic array: an array that grows and shrinks automatically in response to the number of elements it stores.*

`vectors` provide fast element insertion and deletion (amortized constant time) at the end of the `vector`. Amortized constant time insertion means that most of the time insertions are done in constant time $O(1)$ (Chapter 4 explains big-O notation). However, sometimes the `vector` needs to grow in size to accommodate new elements, which has a complexity of $O(N)$. On average this results in $O(1)$ complexity or amortized constant time. Details are explained in Chapter 16. A `vector` has slow (linear time) insertion and deletion anywhere else, because the operation must move all the elements “down” or “up” by one to make room for the new element or to fill the space left by the deleted element. Like arrays, `vectors` provide fast (constant time) access to any of their elements.

You should use a `vector` in your programs when you need fast access to the elements, but do not plan to often add or remove elements in the middle. A good rule of thumb is to use a `vector` whenever you would have used an array. For example, a system-monitoring tool might keep a list of computer systems that it monitors in a `vector`. Only rarely would new computers be added to the list, or current computers removed from the list. However, users would often want to look up information about a particular computer, so lookup times should be fast.

NOTE *Use a vector instead of a C-style array whenever possible.*

There is a template specialization available for `vector<bool>` to store Boolean values in a `vector`. This specialization optimizes space allocation for the Boolean elements; however, the standard does not specify how an implementation of `vector<bool>` should optimize space. The difference between the `vector<bool>` specialization and the `bitset` discussed further in this chapter is that the `bitset` container is of fixed size.

list

An STL `list` is a *doubly linked list* structure and is defined in `<list>`. Like an array or `vector`, it stores a sequence of elements. However, unlike an array or `vector`, the elements of a `list` are not necessarily in contiguous memory. Instead, each element in the `list` specifies where to find the next and previous elements in the `list` (usually via pointers), hence the name *doubly linked list*.

The performance characteristics of a `list` are the exact opposite of a `vector`. They provide slow (linear time) element lookup and access, but quick (constant time) insertion and deletion of elements once the relevant position has been found. Thus, you should use a `list` when you plan to insert and remove many elements but do not require quick lookup.

forward_list

The `forward_list`, defined in `<forward_list>`, is a *singly linked list*, compared to the `list` container, which is doubly linked. The `forward_list` supports forward iteration only, and requires less memory than a `list`. Like `lists`, `forward_lists` allow constant time insertion and deletion anywhere once the relevant position has been found, and there is no fast random access to elements.

deque

The name `deque` is an abbreviation for a *double-ended queue*, although it behaves more like a `vector` instead of a `queue`, which is discussed later. A `deque`, defined in `<deque>`, provides quick (constant time) element access. It also provides fast (amortized constant time) insertion and deletion at both ends of the sequence, but it provides slow (linear time) insertion and deletion in the middle of the sequence.

You should use a `deque` instead of a `vector` when you need to insert or remove elements from either end of the sequence but still need fast access to all elements. However, this requirement does not apply to many programming problems; in most cases a `vector` or `list` should suffice.

array

The `<array>` header defines `array`, which is a replacement for standard C-style arrays. Sometimes you know the exact number of elements in your container up front and you don't need the flexibility of a `vector` or a `list`, which are able to grow dynamically to accommodate new elements. `array` is perfect for such fixed-sized collections and it does not have the same overhead as `vector`; it's basically a thin wrapper around standard C-style arrays. There are a number of advantages in using `arrays` instead of standard C-style arrays; they always know their own size, and do not automatically get cast to a pointer to avoid certain types of bugs. `arrays` do not provide insertion or deletion; they have a fixed size. The advantage of having a fixed size is that this allows an `array` to be allocated on the stack, rather than always demanding heap access as `vector` does. Access to elements is very fast (constant time), just as with `vectors`.

NOTE *The vector, list, forward_list, deque, and array containers are called sequential containers because they store a sequence of elements.*

queue

The name `queue` comes directly from the definition of the English word *queue*, which means a line of people or objects. The `queue` container is defined in `<queue>` and provides standard *first in, first out* (or *FIFO*) semantics. A `queue` is a container in which you insert elements at one end and take them out at the other end. Both insertion (amortized constant time) and removal (constant time) of elements is quick.

You should use a `queue` structure when you want to model real-life “first-come, first-served” semantics. For example, consider a bank. As customers arrive at the bank, they get in line. As tellers become available, they serve the next customer in line, thus providing “first-come, first-served” behavior. You could implement a bank simulation by storing customer objects in a `queue`. As customers arrive at the bank, they are added to the end of the `queue`. As tellers serve customers, they start with customers at the front of the `queue`.

priority_queue

A `priority_queue`, also defined in `<queue>`, provides `queue` functionality in which each element has a priority. Elements are removed from the `queue` in priority order. In the case of priority ties, the order in which elements are removed is undefined. `priority_queue` insertion and deletion are generally slower than simple `queue` insertion and deletion, because the elements must be reordered to support the priority ordering.

You can use `priority_queue`s to model “queues with exceptions.” For example, in the preceding bank simulation, suppose that customers with business accounts take priority over regular customers. Many real-life banks implement this behavior with two separate lines: one for business customers and one for everyone else. Any customers in the business queue are taken before customers in the other line. However, banks could also provide this behavior with a single line in which business customers move to the front of the line ahead of any non-business customers. In your program, you could use a `priority_queue` in which customers have one of two priorities: business or regular. All business customers would be serviced before all regular customers, but each group would be serviced in first-come, first-served order.

stack

The `<stack>` header defines the `stack` class, which provides standard *first-in, last-out* (*FILO*) semantics, also known as *last-in, first-out* (*LIFO*). Like a `queue`, elements are inserted and removed from the container. However, in a `stack`, the most recent element inserted is the first one removed. The name `stack` derives from a visualization of this structure as a stack of objects in which only the top object is visible. When you add an object to the `stack`, you hide all the objects underneath it.

The `stack` container provides fast (constant time) insertion and removal of elements. You should use the `stack` structure when you want *FILO* semantics. For example, an error-processing tool might want to store errors on a `stack` so that the most recent error is the first one available for a

human administrator to read. Processing errors in a FILO order is often useful because newer errors sometimes obviate older ones.

NOTE *Technically, the queue, priority_queue, and stack containers are container adapters. They are simple interfaces built on top of one of the standard sequential containers vector, list, or deque.*

set and multiset

The `set` template is defined in the `<set>` header file, and, as the name suggests, it is a set of elements, loosely analogous to the notion of a mathematical set: Each element is unique, and there is at most one instance of the element in the set. One difference between the mathematical concept of set, and `set` as implemented in the STL, is that in the STL the elements are kept in an order. The reason for the order is that when the client enumerates the elements, they come out in the ordering imposed by the type's `operator<` or a user-defined comparator. The `set` provides logarithmic insertion, deletion, and lookup. This means insertions and deletions are faster than for a `vector` but slower than for a `list`. Lookups are faster than for a `list`, but slower than for a `vector`.

You should use a `set` when you need the elements to be in an order, have equal amounts of insertion/deletion and lookups, and want to optimize performance for both as much as possible. For example, an inventory-tracking program in a busy bookstore might want to use a `set` to store the books. The list of books in stock must be updated whenever books arrive or are sold, so insertion and deletion should be quick. Customers also need the ability to look for a specific book, so the program should provide fast lookup as well.

NOTE *Use a set instead of a vector or list if you need order and want equal performance for insertion, deletion, and lookup.*

Note that a `set` does not allow duplicate elements. That is, each element in the `set` must be unique. If you want to store duplicate elements, you must use a `multiset`, also defined in the `<set>` header file.

map and multimap

The `<map>` header defines the `map` template, which stores key/value pairs. A `map` keeps its elements in sorted order, based on the key values, not the object values. In all other respects, it is identical to a `set`. You should use a `map` when you want to associate keys and values. For example, in the preceding bookstore example, you might want to store the books in a `map` where the key is the ISBN number of the book and the value is a `Book` object containing detailed information for that specific book.

A `multimap`, also defined in `<map>`, has the same relation to a `map` as a `multiset` does to a `set`. Specifically, a `multimap` is a `map` that allows duplicate keys.

Note that you can use a `map` as an *associative array*. That is, you can use it as an array in which the index can be any type; for example, a `string`.

NOTE *The `set` and `map` containers are called associative containers because they associate keys and values. This term is confusing when applied to sets, because in sets the keys are themselves the values. Both containers sort their elements, so they are called sorted or ordered associative containers.*

Unordered Associative Containers / Hash Tables

The STL supports hash tables, also called unordered associative containers. There are four unordered associative containers:

- `unordered_map`
- `unordered_multimap`
- `unordered_set`
- `unordered_multiset`

The first two are defined in `<unordered_map>`, the other two in `<unordered_set>`. Better names would have been `hash_map`, `hash_set`, and so on. Unfortunately, hash tables were not part of the C++ standard library before C++11, which means a lot of third-party libraries implemented hash tables themselves by using names with a prefix `hash` like `hash_map`. Because of this, the C++ standard committee decided to use the prefix `unordered` instead of `hash` to avoid name clashes.

These unordered associative containers behave similar to their ordered counterparts. An `unordered_map` is similar to a standard `map` except that the standard `map` sorts its elements while the `unordered_map` doesn't sort its elements.

Insertion, deletion, and lookup with these unordered associative containers can be done on average in constant time. In a worst-case scenario it will be in linear time. Lookup of elements in an unordered container can be much faster than with a normal `map` or `set`, especially when there are lots of elements in the container.

Chapter 16 explains how these unordered associative containers work and why they are also called hash tables.

`bitset`

C and C++ programmers commonly store a set of flags in a single `int` or `long`, using one bit for each flag. They set and access these bits with the bitwise operators: `&`, `|`, `^`, `~`, `<<`, and `>>`. The C++ standard library provides a `bitset` class that abstracts this bit field manipulation, so you shouldn't need to use the bit manipulation operators anymore.

The `<bitset>` header file defines the `bitset` container, but this is not a container in the normal sense, in that it does not implement a specific data structure in which you insert and remove elements; they have a fixed size and don't support iterators. You can think of them as a

sequence of Boolean values that you can read and write. However, unlike the normal way this is handled in C programming, the `bitset` is not limited to the size of an `int` or other elementary data types. Thus, you can have a 40-bit `bitset`, or a 213-bit `bitset`. The implementation will use as much storage as it needs to implement N bits when you declare your `bitset` with `bitset<N>`.

Summary of STL Containers

The following table summarizes the containers provided by the STL. It uses the big-O notation introduced in Chapter 4 to present the performance characteristics on a container of N elements. An N/A entry in the table means that the operation is not part of the container semantics.

CONTAINER CLASS NAME	CONTAINER TYPE	INSERTION PERFORMANCE	DELETION PERFORMANCE	LOOKUP PERFORMANCE
<code>vector</code>	Sequential	Amortized $O(1)$ at end; $O(N-p)$ for an insert at position p	$O(1)$ at end; $O(N-p)$ for a delete at position p	$O(1)$
When to Use: Need quick lookup. Don't mind slower insertion/deletion. Whenever you would use a standard C-style array that should dynamically grow/shrink in size.				
<code>list</code>	Sequential	$O(1)$ once you are at the position where to insert the element	$O(1)$ once you are at the position where to delete the element	$O(N)$; Statistically $O(N/2)$
When to Use: Need quick insertion/deletion. Don't mind slower lookup.				
<code>forward_list</code>	Sequential	$O(1)$ once you are at the position where to insert the element	$O(1)$ once you are at the position where to delete the element	$O(N)$; Statistically $O(N/2)$
When to Use: When you need the benefits of a <code>list</code> but require only forward iteration.				
<code>deque</code>	Sequential	Amortized $O(1)$ at beginning or end; $O(\min(N-p, p))$ for an insert at position p	$O(1)$ at beginning or end; $O(\min(N-p, p))$ for a delete at position p	$O(1)$
When to Use: Not usually needed; use a <code>vector</code> or <code>list</code> instead.				

array	Sequential	N/A	N/A	O(1)
When to Use: When you need a fixed-size array to replace a standard C-style array.				
queue	Container Adapter	Depends on the underlying container; O(1) for list, amortized O(1) for deque	Depends on the underlying container; O(1) for list and deque	N/A
When to Use: When you want a FIFO structure				
priority_queue	Container Adapter	Depends on the underlying container; amortized O(log(N)) for vector and deque	Depends on the underlying container; O(log(N)) for vector and deque	N/A
When to Use: When you want a queue with priority.				
stack	Container Adapter	Depends on the underlying container; O(1) for list, amortized O(1) for vector and deque	Depends on the underlying container; O(1) for list, vector and deque	N/A
When to Use: When you want a FILO/LIFO structure.				
set multiset	Sorted Associative	O(log(N))	O(log(N))	O(log(N))
When to Use: When you want a sorted collection of elements with equal lookup, insertion, and deletion times.				
map multimap	Sorted Associative	O(log(N))	O(log(N))	O(log(N))
When to Use: When you want a sorted collection to associate keys with values with equal lookup, insertion, and deletion times.				

continues

(continued)

CONTAINER CLASS NAME	CONTAINER TYPE	INSERTION PERFORMANCE	DELETION PERFORMANCE	LOOKUP PERFORMANCE
unordered_map unordered_multimap	Unordered associative / hash table	Average case $O(1)$; worst case $O(N)$	Average case $O(1)$; worst case $O(N)$	Average case $O(1)$; worst case $O(N)$
When to Use: When you want to associate keys with values with equal lookup, insertion, and deletion times and don't require the elements to be sorted. Performance can be better than with a normal <code>map</code> but that depends on the elements.				
unordered_set unordered_multiset	Unordered associative / hash table	Average case $O(1)$; worst case $O(N)$	Average case $O(1)$; worst case $O(N)$	Average case $O(1)$; worst case $O(N)$
When to Use: When you want a collection of elements with equal lookup, insertion, and deletion times and don't require the elements to be sorted. Performance can be better than with a normal <code>set</code> but that depends on the elements.				
bitset	Special	N/A	N/A	$O(1)$
When to Use: When you want a collection of flags.				

Note that `strings` are technically containers as well. They can be thought of as vectors of characters. Thus, some of the algorithms described in the material that follows also work on `strings`.

NOTE `vector` should be your default container. In practice, insertion and deletion in a `vector` is often faster than in a `list` or `forward_list`. This is because of how memory and caches work on modern CPUs, and because of the fact that for a `list` or `forward_list` you first need to iterate to the position where you want to insert or delete an element. Memory for a `list` or `forward_list` might be fragmented, so iteration is slower than for a `vector`.

STL Algorithms

In addition to containers, the STL provides implementations of many generic algorithms. An *algorithm* is a strategy for performing a particular task, such as sorting or searching. These algorithms are also implemented as templates, so they work on most of the different container types. Note that the algorithms are not generally part of the containers. The STL takes the approach of separating the *data* (containers) from the *functionality* (algorithms). Although this

approach seems counter to the spirit of object-oriented programming, it is necessary in order to support generic programming in the STL. The guiding principle of *orthogonality* maintains that algorithms and containers are independent, with (almost) any algorithm working with (almost) any container.

NOTE *Although the algorithms and containers are theoretically independent, some containers provide certain algorithms in the form of class methods because the generic algorithms do not perform well on those particular containers. For example, sets provide their own `find()` algorithm that is faster than the generic `find()` algorithm. You should use the container-specific method form of the algorithm, if provided, because it is generally more efficient or appropriate for the container at hand.*

Note that the generic algorithms do not work directly on the containers. They use an intermediary called an *iterator*. Each container in the STL provides an iterator that supports traversing the elements in the container in a sequence. The different iterators for the various containers adhere to standard interfaces, so algorithms can perform their work by using iterators without worrying about the underlying container implementation. The `<iterator>` header defines a number of helper functions that return specific iterators for containers:

FUNCTION NAME	FUNCTION SYNOPSIS
<code>begin()</code> <code>end()</code>	Returns a non-const iterator to the first and one past the last element in a sequence.
 <code>cbegin()</code> <code>cend()</code>	Returns a const iterator to the first and one past the last element in a sequence.
 <code>rbegin()</code> <code>rend()</code>	Returns a non-const reverse iterator to the last and one before the first element in a sequence.
 <code>crbegin()</code> <code>crend()</code>	Returns a const reverse iterator to the last and one before the first element in a sequence.

This section gives an overview of what kind of algorithms are available in the STL without giving all the fine points. Consult a Standard Library Reference; for example, <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>, for the exact prototypes of all the algorithms. The following chapters go deeper in on iterators, algorithms, and containers with coding examples.

NOTE *Iterators mediate between algorithms and containers. They provide a standard interface to traverse the elements of a container in sequence, so that any algorithm can work on any container.*

There are approximately 90 algorithms in the STL depending on how you count them. The following sections divide these algorithms into different categories. The algorithms are defined in the `<algorithm>` header file unless otherwise noted. Note that whenever the following algorithms are specified as working on a “sequence” of elements, that sequence is presented to the algorithm via iterators.

NOTE *When examining the list of algorithms, remember that the STL is designed with generality in mind, so it adds generality that might never be used, but which, if required, would be essential. You may not need every algorithm, or need to worry about the more obscure parameters that are there for anticipated generality. It is important only to be aware of what's available in case you ever find it useful.*

Non-Modifying Sequence Algorithms

The non-modifying algorithms are those that look at a sequence of elements and return some information about the elements. As “non-modifying” algorithms, they cannot change the values of elements or the order of elements within the sequence. This category contains three types of algorithms. The following tables list and provide brief summaries of the various non-modifying algorithms. With these algorithms, you should rarely need to write a `for` loop to iterate over a sequence of values.

Search Algorithms

These algorithms do not require the sequence to be sorted.

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>adjacent_find()</code>	Finds the first instance of two consecutive elements that are equal to each other or are equivalent to each other as specified by a predicate.	Linear
<code>find()</code> <code>find_if()</code>	Finds the first element that matches a value or causes a predicate to return <code>true</code> .	Linear
<code>find_first_of()</code>	Like <code>find</code> , except searches for one of several elements at the same time.	Quadratic
<code>find_if_not()</code>	Finds the first element that causes a predicate to return <code>false</code> .	Linear
<code>search()</code> <code>find_end()</code>	Finds the first (<code>search()</code>) or last (<code>find_end()</code>) subsequence in a sequence that matches another sequence or whose elements are equivalent, as specified by a predicate.	Quadratic
<code>search_n()</code>	Finds the first instance of n consecutive elements that are equal to a given value or relate to that value according to a predicate.	Linear

Comparison Algorithms

The following comparison algorithms are provided. None of them require the source sequences to be ordered. All of them have a linear worst-case complexity.

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>equal()</code>	Determines if two sequences are equal by checking if parallel elements are equal or match a predicate.
<code>mismatch()</code>	Returns the first element in each sequence that does not match the element in the same location in the other sequence.
<code>lexicographical_compare()</code>	Compares two sequences to determine their “lexicographical” ordering. Compares each element of the first sequence with its equivalent element in the second. If one element is less than the other, that sequence is lexicographically first. If the elements are equal, compares the next elements in order.

Utility Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>all_of()</code>	Returns <code>true</code> if the predicate returns <code>true</code> for all the elements in the sequence or if the sequence is empty; <code>false</code> otherwise.
<code>any_of()</code>	Returns <code>true</code> if the predicate returns <code>true</code> for at least one element in the sequence; <code>false</code> otherwise.
<code>none_of()</code>	Returns <code>true</code> if the predicate returns <code>false</code> for all the elements in the sequence or if the sequence is empty; <code>false</code> otherwise.
<code>count()</code> <code>count_if()</code>	Counts the number of elements matching a value or that cause a predicate to return <code>true</code> .

Modifying Sequence Algorithms

The modifying algorithms modify some or all of the elements in a sequence. Some of them modify elements *in place*, so that the original sequence changes. Others copy the results to a different sequence so that the original sequence is unchanged. All of them have a linear worst-case complexity. The following table summarizes the modifying algorithms:

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>copy()</code>	Copies elements from one sequence to another.
<code>copy_backward()</code>	
<code>copy_if()</code>	Copies elements for which a predicate returns <code>true</code> from one sequence to another.

continues

(continued)

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>copy_n()</code>	Copies n elements from one sequence to another.
<code>fill()</code>	Sets all elements in the sequence to a new value.
<code>fill_n()</code>	Sets the first n elements in the sequence to a new value.
<code>generate()</code>	Calls a specified function to generate a new value for each element in the sequence.
<code>generate_n()</code>	Calls a specified function to generate a new value for the first n elements in the sequence.
<code>move()</code> <code>move_backward()</code>	Moves elements from one sequence to another. This uses efficient move semantics.
<code>remove()</code> <code>remove_if()</code> <code>remove_copy()</code> <code>remove_copy_if()</code>	Removes elements that match a given value or that cause a predicate to return <code>true</code> , either in place or by copying the results to a different sequence.
<code>replace()</code> <code>replace_if()</code> <code>replace_copy()</code> <code>replace_copy_if()</code>	Replaces all elements matching a value or that cause a predicate to return <code>true</code> with a new element, either in place or by copying the results to a different sequence.
<code>reverse()</code> <code>reverse_copy()</code>	Reverses the order of the elements in the sequence, either in place or by copying the results to a different sequence.
<code>rotate()</code> <code>rotate_copy()</code>	Swaps the first and second “halves” of the sequence, either in place or by copying the results to a different sequence. The two subsequences to be swapped need not be equal in size.
<code>shuffle()</code> <code>random_shuffle()</code>	Shuffles the sequence by randomly reordering the elements. It is possible to specify the properties of the random number generator used for shuffling. <code>random_shuffle()</code> is deprecated since C++14.
<code>transform()</code>	Calls a unary function on each element of a sequence or a binary function on parallel elements of two sequences.
<code>unique()</code> <code>unique_copy()</code>	Removes consecutive duplicates from the sequence, either in place or by copying results to a different sequence.

Operational Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>for_each()</code>	Executes a function on each element in the sequence. This algorithm has a linear complexity and does not require the source sequence to be ordered.

Swap Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>iter_swap()</code>	Swaps two elements or sequences of elements.
<code>swap_ranges()</code>	
<code>swap()</code>	Swaps two values, defined in the <code><utility></code> header.

Partition Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>is_partitioned()</code>	Returns <code>true</code> if all elements for which a predicate returns <code>true</code> are before all elements for which it returns <code>false</code> .	Linear
<code>partition()</code>	Sorts the sequence such that all elements for which a predicate returns <code>true</code> are before all elements for which it returns <code>false</code> , without preserving the original order of the elements within each partition.	Linear
<code>stable_partition()</code>	Sorts the sequence such that all elements for which a predicate returns <code>true</code> are before all elements for which it returns <code>false</code> , while preserving the original order of the elements within each partition.	Linear Logarithmic
<code>partition_copy()</code>	Copies elements from one sequence to two different sequences. The target sequence is selected based on the result of a predicate, either <code>true</code> or <code>false</code> .	Linear
<code>partition_point()</code>	Returns an iterator such that all elements before this iterator return <code>true</code> for a predicate and all elements after this iterator return <code>false</code> for that predicate.	Logarithmic

Sorting Algorithms

The STL provides several different sorting algorithms with varying performance guarantees.

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>is_sorted()</code> <code>is_sorted_until()</code>	Checks if a sequence is sorted or which subsequence is sorted.	Linear
<code>nth_element()</code>	Relocates the n th element of the sequence such that the element in the position pointed to by n th is the element that would be in that position if the whole range were sorted, and it rearranges all elements such that all elements preceding the n th element are less than the new n th element, and the ones following it are greater than the new n th element.	Linear
<code>partial_sort()</code> <code>partial_sort_copy()</code>	Partially sorts the sequence: The first n elements (specified by iterators) are sorted; the rest are not. They are sorted either in place or by copying them to a new sequence.	Linear Logarithmic
<code>sort()</code> <code>stable_sort()</code>	Sorts elements in place, either preserving the order of duplicate elements or not.	Linear Logarithmic

Binary Search Algorithms

The following binary search algorithms require the sequence to be at least partitioned on the element that is searched for. This could, for example, be achieved by applying `std::partition()`. A sorted sequence also meets this requirement. All these algorithms have logarithmic complexity.

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>lower_bound()</code> <code>upper_bound()</code> <code>equal_range()</code>	Finds the beginning, (<code>lower_bound()</code>), end (<code>upper_bound()</code>), or both sides (<code>equal_range()</code>) of the range including a specified element.
<code>binary_search()</code>	Finds a value in a sequence.

Set Algorithms

Set algorithms are special modifying algorithms that perform set operations on sequences. They are most appropriate on sequences from set containers, but work on sorted sequences from most containers.

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>inplace_merge()</code>	Merges two sorted sequences in place.	Linear Logarithmic
<code>merge()</code>	Merges two sorted sequences by copying them to a new sequence.	Linear
<code>includes()</code>	Determines if every element from one sequence is in another sequence.	Linear
<code>set_union()</code> <code>set_intersection()</code> <code>set_difference()</code> <code>set_symmetric_difference()</code>	Performs the specified set operation on two sorted sequences, copying results to a third sorted sequence.	Linear

Heap Algorithms

A *heap* is a standard data structure in which the elements of an array or sequence are ordered in a semi-sorted fashion so that finding the “top” element is quick. Six algorithms allow you to work with heaps.

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>is_heap()</code>	Checks if a range of elements is a heap.	Linear
<code>is_heap_until()</code>	Finds the largest subrange in the given range of elements that is a heap.	Linear
<code>make_heap()</code>	Creates a heap from a range of elements.	Linear
<code>push_heap()</code> <code>pop_heap()</code>	Adds or removes an element from the heap.	Logarithmic
<code>sort_heap()</code>	Converts the heap into a range of ascending sorted elements.	Linear Logarithmic

Minimum/Maximum Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>min()</code>	Returns the minimum or maximum of two or more values.
<code>max()</code>	
<code>minmax()</code>	Returns the minimum and maximum of two or more values as a pair.
<code>min_element()</code> <code>max_element()</code>	Finds the minimum or maximum element in a sequence.
<code>minmax_element()</code>	Finds the minimum and maximum element in a sequence and returns the result as a pair.

Numerical Processing Algorithms

The `<numeric>` header provides the following numerical processing algorithms. None of them require the source sequences to be ordered. All of them have a linear complexity.

ALGORITHM NAME	ALGORITHM SYNOPSIS
<code>accumulate()</code>	"Accumulates" the values of all the elements in a sequence. The default behavior is to sum the elements, but the caller can supply a different binary function instead.
<code>adjacent_difference()</code>	Generates a new sequence in which each element is the difference (or other binary operation) of the parallel element, and its predecessor, in the source sequence.
<code>inner_product()</code>	Similar to <code>accumulate()</code> , but works on two sequences. Calls a binary function (multiplication by default) on parallel elements in the sequences, accumulating the result using another binary function (addition by default). If the sequences represent mathematical vectors, the algorithm calculates the dot product of the vectors.
<code>iota()</code>	Fills a sequence with successively incrementing values starting with a given value.
<code>partial_sum()</code>	Generates a new sequence in which each element is the sum (or other binary operation) of the parallel element, and all preceding elements, in the source sequence.

Permutation Algorithms

ALGORITHM NAME	ALGORITHM SYNOPSIS	COMPLEXITY
<code>is_permutation()</code>	Returns <code>true</code> if the elements in one range are a permutation of the elements in another range.	Quadratic
<code>next_permutation()</code> <code>prev_permutation()</code>	Modifies the sequence by transforming it into its lexicographical "next" or "previous" permutation. Successive calls to one or the other will permute the sequence into all possible permutations of its elements if you start with a properly sorted sequence. Returns <code>false</code> if no more permutations exist.	Linear

Choosing an Algorithm

The number and capabilities of the algorithms might overwhelm you at first. It can also be difficult to see how to apply them in the beginning. However, now that you have an idea of the available options, you are better able to tackle your program designs. The next chapters cover the details of how to use these algorithms in your code.

What's Missing from the STL

The STL is powerful, but it's not perfect. Here is a list of omissions and unsupported functionality:

- The STL does not guarantee any thread safety for accessing containers simultaneously from multiple threads.
- The STL does not provide any generic tree or graph structures. Although `maps` and `sets` are generally implemented as balanced binary trees, the STL does not expose this implementation in the interface. If you need a tree or graph structure for something like writing a parser, you will need to implement your own or find an implementation in another library.

It is important to keep in mind that the STL is *extensible*. You can write your own containers or algorithms that will work with existing algorithms or containers. So, if the STL doesn't provide exactly what you need, consider writing your desired code such that it works with the STL. Chapter 20 covers the topic of customizing and extending the STL.

SUMMARY

This chapter provided an overview of the C++ standard library, which is the most important library that you will use in your code. It subsumes the C library and includes additional facilities for strings, I/O, error handling, and other tasks. It also includes generic containers and algorithms, which are together referred to as the standard template library (STL). The next chapters describe the standard template library in more detail.

16

Understanding Containers and Iterators

WHAT'S IN THIS CHAPTER?

- Explaining iterators
- Containers Overview: requirements on elements, general error handling, and iterators
- Sequential Containers: `vector`, `deque`, `list`, `forward_list`, and `array`
- Container Adapters: `queue`, `priority_queue`, and `stack`
- Associative Containers: the `pair` utility, `map`, `multimap`, `set`, and `multiset`
- Unordered Associative Containers/Hash Tables: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`
- Other Containers: standard C-style arrays, strings, streams, and `bitset`

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Chapter 15 introduces the STL, describes its basic philosophy, and provides an overview of the various containers and algorithms. This chapter begins a more-in-depth tour of the STL by covering the STL containers. A detailed list of available classes and methods can be found in a Standard Library Reference; for example, <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>.

The next chapters go deeper on topics such as algorithms, regular expressions, and how you can customize and extend the STL.

CONTAINERS OVERVIEW

Containers in the STL are generic data structures useful for storing collections of data. You should rarely need to use a standard C-style array, write a linked list, or design a stack when you use the STL. The containers are implemented as templates, which allow you to instantiate them for any type that meets certain basic conditions outlined below. Most of the STL containers, except for `array` and `bitset`, are flexible in size and will automatically grow or shrink to accommodate more or fewer elements. This is a huge benefit compared to the old standard C-style arrays, which had a fixed size. Because of the fixed-size nature of standard C-style arrays, they are more vulnerable to overruns, which in the simplest cases merely cause the program to crash because data has been corrupted, but in the worst cases allow certain kinds of security attacks. By using STL containers your programs will be less vulnerable to these kinds of problems.

The STL provides 16 containers, divided into four categories.

- | | |
|--|--|
| <ul style="list-style-type: none">➤ Sequential containers<ul style="list-style-type: none">➤ <code>vector</code> (dynamic array)➤ <code>deque</code>➤ <code>list</code>➤ <code>forward_list</code>➤ <code>array</code>➤ Associative containers<ul style="list-style-type: none">➤ <code>map</code>➤ <code>multimap</code>➤ <code>set</code>➤ <code>multiset</code> | <ul style="list-style-type: none">➤ Unordered associative containers or hash tables<ul style="list-style-type: none">➤ <code>unordered_map</code>➤ <code>unordered_multimap</code>➤ <code>unordered_set</code>➤ <code>unordered_multiset</code>➤ Container adapters<ul style="list-style-type: none">➤ <code>queue</code>➤ <code>priority_queue</code>➤ <code>stack</code> |
|--|--|

Additionally, C++ strings and streams can also be used as STL containers to a certain degree, and `bitset` can be used to store a fixed number of bits.

Everything in the STL is in the `std` namespace. The examples in this book usually use the blanket `using namespace std;` statement in source files (never use this in header files!), but you can be more selective in your own programs about which symbols from `std` to use.

Requirements on Elements

STL containers use value semantics on elements. That is, they store a copy of elements that they are given, assign to elements with the assignment operator, and destroy elements with the destructor.

Thus, when you write classes that you intend to use with the STL, make sure they are copyable. When requesting an element from the container, a reference to the stored copy is returned.

If you prefer reference semantics, you must implement them yourself by storing pointers to elements instead of the elements themselves. When the containers copy a pointer, the result still refers to the same element.

WARNING *If you want to store pointers in containers use `unique_ptr` if the container becomes owner of the pointed-to object, or `shared_ptr` if the container shares ownership with other owners. Do not use the old deprecated `auto_ptr` class in containers because it does not implement copying correctly (as far as the STL is concerned).*

One of the template type parameters for STL containers is a so-called allocator. The container can use this allocator to allocate and deallocate memory for elements. Some containers, such as a `map`; also accept a comparator as one of the template type parameters. This comparator is used to order elements. Both of these template parameters have default values.

The specific requirements on elements in containers using the default allocator and comparator are shown in the following table:

METHOD	DESCRIPTION	NOTES
Copy Constructor	Creates a new element that is “equal” to the old one, but that can safely be destructed without affecting the old one	Used every time you insert an element, except when using an <code>emplace</code> method
Move Constructor	Creates a new element by moving all content from a source element to the new element	Used when the source element is an <code>rvalue</code> , and will be destroyed after the construction of the new element; also used when a <code>vector</code> grows in size, for example
Assignment Operator	Replaces the contents of an element with a copy of the source element	Used every time you modify an element
Move Assignment Operator	Replaces the contents of an element by moving all content from a source element	Used when the source element is an <code>rvalue</code> , and will be destroyed after the assignment operation
Destructor	Cleans up an element	Used every time you remove an element, or, for example, when a <code>vector</code> grows in size and the elements are not movable

continues

(continued)

METHOD	DESCRIPTION	NOTES
Default Constructor	Constructs an element without any arguments	Required only for certain operations, such as the <code>vector::resize()</code> method with one argument, and the <code>map::operator[]</code> access
<code>operator==</code>	Compares two elements for equality	Required for keys in unordered containers, and for certain operations, such as <code>operator==</code> on two containers
<code>operator<</code>	Determines if one element is less than another	Required for keys in associative containers, and for certain operations, such as <code>operator<</code> on two containers

Chapter 8 shows you how to write these methods. Move semantics is discussed in Chapter 10. For move semantics to work properly with STL containers, the move constructor and the move assignment operator must be marked as `noexcept`.

WARNING *The STL containers call the copy constructor and assignment operator for elements often, so make those operations efficient. You can also increase performance by implementing move semantics for your elements, as described in Chapter 10.*

Exceptions and Error Checking

The STL containers provide limited error checking. Clients are expected to ensure that their uses are valid. However, some container methods and functions throw exceptions in certain conditions such as out-of-bounds indexing. This chapter mentions exceptions where appropriate. Consult a Standard Library Reference for a list of possible exceptions thrown from each method. However, it is impossible to list exhaustively the exceptions that can be thrown from these methods because they perform operations on user-specified types with unknown exception characteristics.

Iterators

The STL uses the iterator pattern to provide a generic abstraction for accessing the elements of the containers. Each container provides a container-specific iterator, which is a glorified smart pointer that knows how to iterate over the elements of that specific container. The iterators for all the different containers adhere to a specific interface defined in the C++ standard. Thus, even though the containers provide different functionality, the iterators present a common interface to code that wishes to work with elements of the containers.

You can think of an iterator as a pointer to a specific element of the container. Like pointers to elements in an array, iterators can move to the next element with `operator++`. Similarly, you can usually use `operator*` and `operator->` on the iterator to access the actual element or field of the element. Some iterators allow comparison with `operator==` and `operator!=`, and support `operator--` for moving to previous elements.

All iterators must be copy constructible, copy assignable, and destructible. Lvalues of iterators must be swappable. Different containers provide iterators with slightly different additional capabilities. The standard defines five categories of iterators, summarized in the following table.

ITERATOR CATEGORY	OPERATIONS REQUIRED	COMMENTS
Read (officially called "input" iterator)	<code>operator++</code> <code>operator*</code> <code>operator-></code> <code>copy constructor</code> <code>operator=</code> <code>operator==</code> <code>operator!=</code>	Provides read-only access, forward only (no <code>operator--</code> to move backward). Iterators can be assigned, copied, and compared for equality.
Write (officially called "output" iterator)	<code>operator++</code> <code>operator*</code> <code>copy constructor</code> <code>operator=</code>	Provides write-only access, forward only. Iterators can be assigned, but cannot be compared for equality. Note the absence of <code>operator-></code> .
Forward	Capabilities of input iterators, plus: default constructor	Provides read/write access, forward only. Iterators can be assigned, copied, and compared for equality.
Bidirectional	Capabilities of forward iterators, plus: <code>operator--</code>	Provides everything forward iterator provides. Iterators can also move backward to previous element.
Random Access	Bidirectional capability, plus: <code>operator+</code> <code>operator-</code> <code>operator+=</code> <code>operator-=</code> <code>operator<</code> <code>operator></code> <code>operator<=</code> <code>operator>=</code> <code>operator[]</code>	Equivalent to dumb pointers: Iterators support pointer arithmetic, array index syntax, and all forms of comparison.

Additionally, you can use `std::distance()` to compute the distance between two iterators of a container.

Iterators are implemented similarly to smart pointer classes in that they overload the specific desired operators. Consult Chapter 14 for details on operator overloading.

The basic iterator operations are similar to those supported by dumb pointers, so a dumb pointer can be a legitimate iterator for certain containers. In fact, the `vector` iterator could technically be implemented as a simple dumb pointer. However, as a client of the containers, you need not worry about the implementation details; you can simply use the iterator abstraction.

NOTE *Iterators might not be implemented internally as pointers, so this text uses the term “refers to” instead of “points to” when discussing the elements accessible via an iterator.*

Chapters 17 delves into more detail about iterators and the STL algorithms that use them. This chapter shows you the basics of using the iterators for each container.

NOTE *Only the sequential containers, associative containers, and unordered associative containers provide iterators. The container adapters and bitset class do not support iteration over their elements.*

Common Iterator `typedefs` and Methods

Every container class in the STL that supports iterators provides public `typedefs` for its iterator types called `iterator` and `const_iterator`. For example, a `const` iterator for a `vector` of `ints` has as type `std::vector<int>::const_iterator`. Containers that allow you to iterate over its elements in reverse order also provide public `typedefs` called `reverse_iterator` and `const_reverse_iterator`. This way, clients can use the container iterators without worrying about the actual types.

NOTE *const_iterators and const_reverse_iterators provide read-only access to elements of the container.*

The containers also provide a method `begin()` that returns an iterator referring to the first element in the container. The `end()` method returns an iterator to the “past-the-end” value of the sequence of elements. That is, `end()` returns an iterator that is equal to the result of applying `operator++` to an iterator referring to the last element in the sequence. Together `begin()` and `end()` provide a *half-open range* that includes the first element but not the last. The reason for this apparent complication is to support empty ranges (containers without any elements), in which case `begin()` is equal to `end()`. The half-open range bounded by iterators `begin()` and `end()` is often written mathematically like this: `[begin,end)`.

NOTE *The half-open range concept also applies to iterator ranges that are passed to container methods such as `insert()` and `erase()`. See the specific container descriptions later in this chapter for details.*

Similarly, there are:

- `cbegin()` and `cend()` methods that return `const` iterators.
- `rbegin()` and `rend()` methods that return reverse iterators.
- `crbegin()` and `crend()` methods that return `const` reverse iterators.

NOTE *The standard library also provides global non-member functions called `std::begin()`, and `end()`, while C++14 adds `std::cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, and `crend()`. It's recommended to use these non-member functions instead of the member versions.*

SEQUENTIAL CONTAINERS

`vector`, `deque`, `list`, `forward_list`, and `array` are called *sequential containers*. The best way to learn about sequential containers is to jump in with an example of the `vector` container, which is the container most commonly used. The next section describes the `vector` container in detail, followed by briefer descriptions of `deque`, `list`, `forward_list`, and `array`. Once you become familiar with the sequential containers, it's trivial to switch between them.

vector

The STL `vector` container is similar to a standard C-style array: the elements are stored in contiguous memory, each in its own “slot.” You can index into a `vector`, as well as add new elements to the back or insert them anywhere else. Inserting and deleting elements into and from a `vector` generally takes linear time, though these operations actually run in *amortized constant* time at the end of a `vector`, explained in the section “The `vector` Memory Allocation Scheme” later in this chapter. Random access of individual elements has a constant complexity.

vector Overview

`vector` is defined in the `<vector>` header file as a class template with two type parameters: the element type to store and an *allocator* type.

```
template <class T, class Allocator = allocator<T> > class vector;
```

The `Allocator` parameter specifies the type for a memory allocator object that the client can set in order to use custom memory allocation. This template parameter has a default value.

NOTE *The default value for the Allocator type parameter is sufficient for most applications. This chapter assumes that you always use the default allocator. Chapter 20 provides more details in case you are interested.*

Fixed-Length vectors

The simplest way to use a `vector` is as a fixed-length array. `vector` provides a constructor that allows you to specify the number of elements, and provides an overloaded `operator[]` in order to access and modify those elements. The C++ standard states that the result of `operator[]` is undefined when used to access an element outside the `vector` bounds. This means that any compiler can decide how to behave in that case. For example, the default behavior of Microsoft Visual C++ is to give a run-time error message when your program is compiled in debug mode, and to disable any bounds checking in release mode for performance reasons. You can change these default behaviors.

WARNING *Like “real” array indexing, the `operator[]` on a `vector` does not provide bounds checking.*

In addition to using `operator[]`, you can access `vector` elements via `at()`, `front()`, and `back()`. The `at()` method is identical to `operator[]`, except that it performs bounds checking, and throws an `out_of_range` exception if the index is out of bounds. `front()` and `back()` return references to the first and last elements of a `vector`, respectively. Calling `front()` or `back()` on an empty container causes undefined behavior.

NOTE *All vector element accesses run with constant complexity.*

Here is a small example program to “normalize” test scores so that the highest score is set to 100, and all other scores are adjusted accordingly. The program creates a `vector` of 10 `doubles`, reads in 10 values from the user, divides each value by the max score (times 100), and prints out the new values. For the sake of brevity, the program forsakes error checking.

```
vector<double> doubleVector(10); // Create a vector of 10 doubles.
// Initialize max to smallest number
double max = numeric_limits<double>::infinity();
for (size_t i = 0; i < doubleVector.size(); i++) {
    cout << "Enter score " << i + 1 << ": ";
    cin >> doubleVector[i];
    if (doubleVector[i] > max) {
        max = doubleVector[i];
    }
}
max /= 100.0;
for (auto& element : doubleVector) {
    element /= max;
    cout << element << " ";
}
```

As you can see from this example, you can use a `vector` just as you would use a standard C-style array. Note that the first `for` loop uses the `size()` method to determine the number of elements in the container. The example also demonstrates the use of a range-based `for` loop with a `vector`. In this example, the range-based `for` loop uses `auto&` and not `auto` because a reference is required so that the element can be modified in each iteration.

NOTE *The operator [] on a vector normally returns a reference to the element, which can be used on the left-hand side of assignment statements. If operator[] is called on a const vector object, it returns a reference to a const element, which cannot be used as the target of an assignment. See Chapter 14 for details on how this trick is implemented.*

Dynamic-Length vectors

The real power of a `vector` lies in its ability to grow dynamically. For example, consider the test score normalization program from the previous section with the additional requirement that it should handle any number of test scores. Here is the new version:

```
vector<double> doubleVector; // Create a vector with zero elements.
// Initialize max to smallest number
double max = -numeric_limits<double>::infinity();
for (size_t i = 1; true; i++) {
    double temp;
    cout << "Enter score " << i << " (-1 to stop): ";
    cin >> temp;
    if (temp == -1) {
        break;
    }
    doubleVector.push_back(temp);
    if (temp > max) {
        max = temp;
    }
}
max /= 100.0;
for (auto& element : doubleVector) {
    element /= max;
    cout << element << " ";
}
```

This version of the program uses the default constructor to create a `vector` with zero elements. As each score is read, it's added to the `vector` with the `push_back()` method, which takes care of allocating space for the new element. The range-based `for` loop doesn't require any changes.

vector Details

Now that you've had a taste of `vectors`, it's time to delve into their details.

Constructors and Destructors

The default constructor creates a `vector` with 0 elements.

```
vector<int> intVector; // Creates a vector of ints with zero elements
```

You can specify a number of elements and, optionally, a value for those elements, like this:

```
vector<int> intVector(10, 100); // Creates vector of 10 ints with value 100
```

If you omit the default value, the new objects are zero-initialized. Zero-initialization constructs objects with the default constructor, initializes primitive integer types (such as `char`, `int`, etc.) to 0, primitive floating point types to 0.0, and pointer types to `nullptr`.

You can create vectors of built-in classes like this:

```
vector<string> stringVector(10, "hello");
```

User-defined classes can also be used as vector elements:

```
class Element
{
public:
    Element() {}
    virtual ~Element() {}
};

...
vector<Element> elementVector;
```

A vector can be constructed with an `initializer_list` containing the initial elements:

```
vector<int> intVector({ 1, 2, 3, 4, 5, 6 });
```

`initializer_lists` can also be used for so-called *uniform initialization*, as discussed in Chapter 10. Uniform initialization works on most STL containers. For example:

```
vector<int> intVector1 = { 1, 2, 3, 4, 5, 6 };
vector<int> intVector2{ 1, 2, 3, 4, 5, 6 };
```

You can allocate vectors on the heap as well:

```
auto elementVector = make_unique<vector<Element>>(10);
```

Copying and Assigning vectors

A vector stores copies of the objects, and its destructor calls the destructor for each of the objects. The copy constructor and assignment operator of the `vector` class perform deep copies of all the elements in the vector. Thus, for efficiency, you should pass vectors by reference or `const` reference to functions and methods. Consult Chapter 11 for the details on writing functions that take template instantiations as parameters.

In addition to normal copying and assignment, vectors provide an `assign()` method that removes all the current elements and adds any number of new elements. This method is useful if you want to reuse a vector. Here is a trivial example. `intVector` is created with 10 elements having the default value 0. Then `assign()` is used to remove all 10 elements and replace them with 5 elements with value 100.

```
vector<int> intVector(10);
// Other code . . .
intVector.assign(5, 100);
```

`assign()` can also accept an `initializer_list` as follows. `intVector` will now have 4 elements with the given values.

```
intVector.assign({ 1, 2, 3, 4 });
```

vectors also provide a `swap()` method that allows you to swap the contents of two vectors in constant time. Here is a simple example:

```
vector<int> vectorOne(10);
vector<int> vectorTwo(5, 100);
vectorOne.swap(vectorTwo);
// vectorOne now has 5 elements with the value 100.
// vectorTwo now has 10 elements with the value 0.
```

Comparing vectors

The STL provides the usual six overloaded comparison operators for vectors: `==`, `!=`, `<`, `>`, `<=`, `>=`. Two vectors are equal if they have the same number of elements and all the corresponding elements in the two vectors are equal to each other. Two vectors are compared lexicographically, that is, one vector is “less than” another if all elements 0 through $i-1$ in the first vector are equal to elements 0 through $i-1$ in the second vector, but element i in the first is less than element i in the second, where i must be in the range $0 \dots n$ and n must be less than `size()`.

NOTE Comparing two vectors with operator`==` or operator`!=` requires the individual elements to be comparable with operator`==`. Comparing two vectors with operator`<`, operator`>`, operator`<=`, or operator`>=` requires the individual elements to be comparable with operator`<`. If you intend to store objects of a custom class in a vector, make sure to write those operators.

Here is an example of a simple program that compares vectors of ints:

```
vector<int> vectorOne(10);
vector<int> vectorTwo(10);
if (vectorOne == vectorTwo) {
    cout << "equal!" << endl;
} else {
    cout << "not equal!" << endl;
}
vectorOne[3] = 50;
if (vectorOne < vectorTwo) {
    cout << "vectorOne is less than vectorTwo" << endl;
} else {
    cout << "vectorOne is not less than vectorTwo" << endl;
}
```

The output of the program is as follows:

```
equal!
vectorOne is not less than vectorTwo
```

vector Iterators

The section on “Iterators” at the beginning of this chapter explained the basics of container iterators. The discussion can get a bit abstract, so it’s helpful to jump in and look at a code example. Here is the test score normalization program from earlier with the range-based `for` loop replaced by a `for` loop using an iterator:

```
vector<double> doubleVector;
// Initialize max to smallest number
double max = numeric_limits<double>::infinity();
for (size_t i = 1; true; i++) {
    double temp;
    cout << "Enter score " << i << " (-1 to stop): ";
    cin >> temp;
    if (temp == -1) {
        break;
    }
    doubleVector.push_back(temp);
    if (temp > max) {
        max = temp;
    }
}
max /= 100.0;
for (vector<double>::iterator iter = begin(doubleVector);
     iter != end(doubleVector); ++iter) {
    *iter /= max;
    cout << *iter << " ";
}
```

First, take a look at the `for` loop initialization statement:

```
vector<double>::iterator iter = begin(doubleVector);
```

Recall that every container defines a type named `iterator` to represent iterators for that type of container. `begin()` returns an iterator of that type referring to the first element in the container. Thus, the initialization statement obtains in the variable `iter` an iterator referring to the first element of `doubleVector`. Next, look at the `for` loop comparison:

```
iter != end(doubleVector);
```

This statement simply checks if the iterator is past the end of the sequence of elements in the `vector`. When it reaches that point, the loop terminates. The increment statement, `++iter`, increments the iterator to refer to the next element in the `vector`.

NOTE Use pre-increment instead of post-increment when possible because pre-increment is at least as efficient, and usually more efficient. `iter++` must return a new iterator object, while `++iter` can simply return a reference to `iter`. See Chapter 14 for details on implementing both versions of `operator++`.

The `for` loop body contains these two lines:

```
*iter /= max;
cout << *iter << " ";
```

As you can see, your code can both access and modify the elements over which it iterates. The first line uses `*` to dereference `iter` to obtain the element to which it refers, and assigns to that element. The second line dereferences `iter` again, but this time only to stream the element to `cout`.

The preceding `for` loop using iterators can be simplified by using the `auto` keyword:

```
for (auto iter = begin(doubleVector);
     iter != end(doubleVector); ++iter) {
    *iter /= max;
    cout << *iter << " ";
```

In this example, the compiler will automatically deduce the type of the variable `iter` based on the right-hand side of the initializer, which in this case is the result of the call to `begin()`.

Accessing Fields of Object Elements

If the elements of your container are objects, you can use the `->` operator on iterators to call methods or access data members of those objects. For example, the following program creates a `vector` of 10 `string`s, then iterates over all of them appending a new `string` to the old one:

```
vector<string> stringVector(10, "hello");
for (auto it = begin(stringVector); it != end(stringVector); ++it) {
    it->append(" there");
}
```

Or, using a range-based `for` loop, it can be written as follows:

```
vector<string> stringVector(10, "hello");
for (auto& str : stringVector) {
    str.append(" there");
}
```

const_iterator

The normal iterator is read/write. However, if you call `begin()` or `end()` on a `const` object, or you call `cbegin()` or `cend()`, you receive a `const_iterator`. The `const_iterator` is read-only; you cannot modify the elements. An iterator can always be converted to a `const_iterator`, so it's always safe to write something like this:

```
vector<type>::const_iterator it = begin(myVector);
```

However, a `const_iterator` cannot be converted to an `iterator`. If `myVector` is `const`, the following line doesn't compile:

```
vector<type>::iterator it = begin(myVector);
```

NOTE *If you do not need to modify the elements of a vector, you should use a `const_iterator`. This rule will make it easier to guarantee correctness of your code and allows compilers to perform certain optimizations.*

When using the `auto` keyword, using `const_iterators` looks a bit different. Suppose you write the following code:

```
vector<string> stringVector(10, "hello");
for (auto iter = begin(stringVector); iter != end(stringVector); ++iter) {
    cout << *iter << endl;
}
```

Because of the `auto` keyword, the compiler will deduce the type of the `iter` variable automatically and will make it a normal iterator, meaning that you can read and write to the iterator. If you want a read-only `const_iterator` in combination with using `auto`, then you need to use `cbegin()` and `cend()` instead of `begin()` and `end()` as follows:

```
vector<string> stringVector(10, "hello");
for (auto iter = cbegin(stringVector); iter != cend(stringVector); ++iter) {
    cout << *iter << endl;
}
```

Now the compiler will use the `const_iterator` as type for the variable `iter` because that's what `cbegin()` returns.

NOTE *The non-member functions `cbegin()` and `cend()` are available since C++14. In C++11 you should use the `cbegin()` and `cend()` member functions.*

A range-based `for` loop can also be forced to use `const` iterators as follows:

```
vector<string> stringVector(10, "hello");
for (const auto& element : stringVector) {
    cout << element << endl;
}
```

Iterator Safety

Generally, iterators are about as safe as pointers: extremely insecure. For example, you can write code like this:

```
vector<int> intVector;
auto iter = end(intVector);
*iter = 10; // BUG! iter doesn't refer to a valid element.
```

Recall that the iterator returned by `end()` is past the end of a vector, not an iterator referring to the last element. Trying to dereference it results in undefined behavior. Iterators are not required to perform any verification.

Another problem can occur if you use mismatched iterators. For example, the following code initializes an iterator from `vectorTwo` and tries to compare it to the end iterator for `vectorOne`. Needless to say, this loop will not do what you intended, and may never terminate. Dereferencing the iterator in the loop will likely produce undefined results.

```
vector<int> vectorOne(10);
vector<int> vectorTwo(10);
// Fill in the vectors.
// BUG! Infinite loop
```

```
for (auto iter = begin(vectorTwo); iter != end(vectorOne); ++iter) {
    // Loop body
}
```

NOTE Microsoft Visual C++ by default gives an assertion error at run time for both of the preceding problems when running a debug build of your program. By default, no verification of iterators is performed for release builds. You can enable it for release builds as well, but it has a performance penalty.

Other Iterator Operations

The vector iterator is random access, which means that you can move it backward or forward, or jump around. For example, the following code eventually changes the fifth element (index 4) to the value 4:

```
vector<int> intVector(10);
auto it = begin(intVector);
it += 5;
--it;
*it = 4;
```

Iterators versus Indexing

Given that you can write a `for` loop that uses a simple index variable and the `size()` method to iterate over the elements of the `vector`, why should you bother using iterators? That's a valid question, for which there are three main answers:

- Iterators allow you to insert and delete elements and sequences of elements at any point in the container. See the following “Adding and Removing Elements” section.
- Iterators allow you to use the STL algorithms, which are discussed in Chapter 17.
- Using an iterator to access each element sequentially is often more efficient than indexing the container to retrieve each element individually. This generalization is not true for `vectors`, but applies to `lists`, `maps`, and `sets`.

Adding and Removing Elements

As you have already read, you can append an element to a `vector` with the `push_back()` method. The `vector` provides a parallel remove method called `pop_back()`.

WARNING `pop_back()` does not return the element that it removed. If you want the element you must first retrieve it with `back()`.

You can also insert elements at any point in the `vector` with the `insert()` method, which adds one or more elements to a position specified by an iterator, shifting all subsequent elements down to make room for the new ones. There are five different overloaded forms of `insert()` that:

- insert a single element.
- insert n copies of a single element.

- insert elements from an iterator range. Recall that the iterator range is half-open, such that it includes the element referred to by the starting iterator but not the one referred to by the ending iterator.
- insert a single element by moving the given element to a vector using move semantics.
- insert a list of elements into a vector where the list of elements is given as an `initializer_list`.

NOTE *There are versions of `push_back()` and `insert()` that take an lvalue or an rvalue as parameter. The lvalue versions allocate memory as needed to store the new elements, and store copies of the element arguments. The rvalue versions use move semantics to move ownership of the object to a vector instead of copying the object.*

You can remove elements from any point in a vector with `erase()` and you can remove all elements with `clear()`. There are two forms of `erase()`: one accepting a single iterator to remove a single element, and one accepting two iterators specifying a range of elements to remove.

If you want to remove a number of elements that satisfy a certain condition, one solution would be to write a loop iterating over all the elements and erasing every element that matches the condition. However, this solution has quadratic complexity, which is very bad for performance. In this case, the quadratic complexity can be avoided by using the *remove-erase-idiom*, which has a linear complexity. The remove-erase-idiom is discussed in Chapter 17.

Here is a small program that demonstrates the methods for adding and removing elements. It uses a helper function `printVector()` that prints the contents of a vector to `cout`. The example includes demonstrations of the two-argument version of `erase()` and the following versions of `insert()`:

- `insert(const_iterator pos, const T& x)`: the value `x` will be inserted at position `pos`.
- `insert(const_iterator pos, size_type n, const T& x)`: the value `x` will be inserted `n` times at position `pos`.
- `insert(const_iterator pos, InputIterator first, InputIterator last)`: the elements in the range `first, last` are inserted at position `pos`.

```
vector<int> vectorOne = { 1, 2, 3, 5 };
vector<int> vectorTwo;
// Oops, we forgot to add 4. Insert it in the correct place
vectorOne.insert(cbegin(vectorOne) + 3, 4);
// Add elements 6 through 10 to vectorTwo
for (int i = 6; i <= 10; i++) {
    vectorTwo.push_back(i);
}
printVector(vectorOne);
printVector(vectorTwo);
// Add all the elements from vectorTwo to the end of vectorOne
vectorOne.insert(cend(vectorOne), cbegin(vectorTwo), cend(vectorTwo));
printVector(vectorOne);
// Now erase the numbers 2 through 5 in vectorOne
```

```

vectorOne.erase(cbegin(vectorOne) + 1, cbegin(vectorOne) + 5);
printVector(vectorOne);
// Clear vectorTwo entirely
vectorTwo.clear();
// And add 10 copies of the value 100
vectorTwo.insert(cbegin(vectorTwo), 10, 100);
// Decide we only want 9 elements
vectorTwo.pop_back();
printVector(vectorTwo);

```

The output of the program is as follows:

```

1 2 3 4 5
6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 6 7 8 9 10
100 100 100 100 100 100 100 100 100 100

```

Recall that iterator pairs represent a half-open range, and `insert()` adds elements before the element referred to by the iterator position. Thus, you can insert the entire contents of `vectorTwo` into the end of `vectorOne`, like this:

```
vectorOne.insert(cend(vectorOne), cbegin(vectorTwo), cend(vectorTwo));
```

WARNING *Methods such as `insert()` and `erase()` that take a vector range as arguments assume that the beginning and ending iterators refer to elements in the same container, and that the end iterator refers to an element at or past the begin iterator. The methods will not work correctly if these preconditions are not met!*

Move Semantics

All STL containers implement move semantics by including a move constructor and move assignment operator. These use rvalue references, as described in Chapter 10. A big benefit of this is that you can easily return an STL container from a function *by value* without performance degradation. Take a look at the following function:

```

vector<int> createVectorOfSize(size_t size)
{
    vector<int> vec(size);
    int contents = 0;
    for (auto& i : vec) {
        i = contents++;
    }
    return vec;
}
...
vector<int> myVector;
myVector = createVectorOfSize(123);

```

Without move semantics, `createVectorOfSize()` creates a local `vector` called `vec`. The return statement then makes a *copy* of `vec`, returns it from the function, and assigns it to `myVector`. With

the move semantics support in the STL containers, this copying of the vector is avoided. Instead, the return statement *moves* the vector. Moving is possible in this case because `vec` will go out of scope.

Similarly, push operations can also make use of move semantics to improve performance in certain situations. For example, suppose you have a vector of elements of type `Element` as follows:

```
class Element
{
public:
    Element(int i, const string& str) : mI(i), mStr(str) {}
private:
    int mI;
    string mStr;
};

vector<Element> vec;
```

Adding an element to this vector can be done as follows:

```
Element myElement(12, "Twelve");
vec.push_back(myElement);
```

However, since `myElement` is not a temporary object, `push_back()` makes a copy of `myElement` and puts it in the vector. This copying can be avoided if you call the `push_back()` method as follows:

```
vec.push_back(move(myElement));
```

Now you are explicitly saying that `myElement` should be moved into the vector. Note that after this call, you should not use `myElement` anymore! You can also call `push_back()` as follows:

```
vec.push_back(Element(12, "Twelve"));
```

The `vector` class defines a `push_back(T&& val)`, which is the move equivalent of `push_back(const T& val)`. The preceding call to `push_back()` triggers a call to the move version because the call to the `Element` constructor results in a temporary object. The `push_back()` method moves this temporary `Element` object into the vector, avoiding any copying.

Using uniform initialization the preceding can also be written as follows:

```
vec.push_back({12, "Twelve"});
```

Emplace Operations

C++ supports *emplace* operations on most STL containers, including `vector`. Emplace means “to put into place.” An example is `emplace_back()` on a `vector` object, which does not copy or move anything. Instead, it makes space in the container and constructs the object *in place*. For example:

```
vec.emplace_back(12, "Twelve");
```

The `emplace` methods take a variable number of arguments as a variadic template. Variadic templates are discussed in Chapter 21, but those details are not required to understand how to use `emplace_back()`. The difference in performance between `emplace_back()` and `push_back()`

using move semantics depends on how your specific compiler implements these operations. In most situations you can pick the one based on the syntax that you prefer.

```
vec.push_back({12, "Twelve"});
// Or
vec.emplace_back(12, "Twelve");
```

There is also an `emplace()` method that constructs an object in place at a specific position in the `vector`.

Algorithmic Complexity and Iterator Invalidation

Inserting or erasing elements in a `vector` causes all subsequent elements to shift up or down to make room for, or fill in the holes left by, the affected elements. Thus, these operations take linear complexity. Furthermore, all iterators referring to the insertion or removal point or subsequent positions are invalid following the action. The iterators are not “magically” moved to keep up with the elements that are shifted up or down in the `vector` — that’s up to you.

Also keep in mind that an internal `vector` reallocation can cause invalidation of all iterators referring to elements in the `vector`, not just those referring to elements past the point of insertion or deletion. See the next section for details.

The `vector` Memory Allocation Scheme

A `vector` allocates memory automatically to store the elements that you insert. Recall that the `vector` requirements dictate that the elements must be in contiguous memory, like in standard C-style arrays. Because it’s impossible to request to add memory to the end of a current chunk of memory, every time a `vector` allocates more memory it must allocate a new, larger chunk in a separate memory location and copy/move all the elements to the new chunk. This process is time-consuming, so `vector` implementations attempt to avoid it by allocating more space than needed when they have to perform a reallocation. That way, they can avoid reallocating memory every time you insert an element.

One obvious question at this point is why you, as a client of `vector`, care how it manages its memory internally. You might think that the principle of abstraction should allow you to disregard the internals of the `vector` memory allocation scheme. Unfortunately, there are two reasons why you need to understand how it works:

- 1. Efficiency.** The `vector` allocation scheme can guarantee that an element insert runs in *amortized constant time*: Most of the time the operation is constant, but once in a while (if it requires a reallocation), it’s linear. If you are worried about efficiency you can control when a `vector` performs reallocations.
- 2. Iterator invalidations.** A reallocation invalidates all iterators referring to elements in a `vector`.

Thus, the `vector` interface allows you to query and control the `vector` reallocations. If you don’t control the reallocations explicitly, you should assume that all insertions cause a reallocation and thus invalidate all iterators.

Size and Capacity

`vector` provides two methods for obtaining information about its size: `size()` and `capacity()`. The `size()` method returns the number of elements in a `vector`, while `capacity()` returns the number of elements that it can hold without a reallocation. Thus, the number of elements that you can insert without causing a reallocation is `capacity() - size()`.

NOTE *You can query whether a vector is empty with the `empty()` method. A vector can be empty but have nonzero capacity.*

Reserving Capacity

If you don't care about efficiency or iterator invalidations, there is never a need to control the `vector` memory allocation explicitly. However, if you want to make your program as efficient as possible, or want to guarantee that iterators will not be invalidated, you can force a `vector` to preallocate enough space to hold all of its elements. Of course, you need to know how many elements it will hold, which is sometimes impossible to predict.

One way to preallocate space is to call `reserve()`, which allocates enough memory to hold the specified number of elements. The next section shows an example of the `reserve()` method in action.

WARNING *Reserving space for elements changes the capacity, but not the size. That is, it doesn't actually create elements. Don't access elements past a vector's size.*

Another way to preallocate space is to specify, in the constructor or with the `resize()` method, how many elements you want a `vector` to store. This method actually creates a `vector` of that size (and probably of that capacity).

`vector` Example: A Round-Robin Class

A common problem in computer science is distributing requests among a finite list of resources. For example, a simple operating system could keep a list of processes and assign a time slice (for example, 100ms) to each process to let the process perform some of its work. After the time slice is finished, the OS suspends the process and the next process in the list is given a time slice to perform some of its work. One of the simplest algorithmic solutions to this problem is *round-robin scheduling*. When the time slice of the last process is finished, the scheduler starts over again with the first process. For example, in the case of three processes, the first time slice would go to the first process, the second to the second process, the third to the third process, and the fourth back to the first process. The cycle would continue in this way indefinitely.

Suppose that you decide to write a generic round-robin scheduling class that can be used with any type of resource. The class should support adding and removing resources, and should support cycling through the resources in order to obtain the next one. You could use a `vector` directly, but it's often helpful to write a wrapper class that provides more directly the functionality you need for

your specific application. The following example shows a `RoundRobin` class template with comments explaining the code. First, here is the class definition:

```

// Class template RoundRobin
// Provides simple round-robin semantics for a list of elements.
template <typename T>
class RoundRobin
{
public:
    // Client can give a hint as to the number of expected elements for
    // increased efficiency.
    RoundRobin(size_t numExpected = 0);
    virtual ~RoundRobin();
    // prevent assignment and pass-by-value
    RoundRobin(const RoundRobin& src) = delete;
    RoundRobin& operator=(const RoundRobin& rhs) = delete;
    // Appends elem to the end of the list. May be called
    // between calls to getNext().
    void add(const T& elem);
    // Removes the first (and only the first) element
    // in the list that is equal (with operator==) to elem.
    // May be called between calls to getNext().
    void remove(const T& elem);
    // Returns the next element in the list, starting with the first,
    // and cycling back to the first when the end of the list is
    // reached, taking into account elements that are added or removed.
    T& getNext();
private:
    std::vector<T> mElems;
    typename std::vector<T>::iterator mCurElem;
};

```

As you can see, the public interface is straightforward: only three methods plus the constructor and destructor. The resources are stored in the vector called `mElems`. The iterator `mCurElem` always refers to the element that was previously returned by a call to `getNext()`. If `getNext()` hasn't been called yet, `mCurElem` is equal to `end(mElems)`. Note the use of the `typename` keyword in front of the line declaring `mCurElem`. So far, you've only seen that keyword used to specify template parameters, but there is another use for it. You must specify `typename` explicitly whenever you access a type based on one or more template parameters. In this case, the template parameter `T` is used to access the iterator type. Thus, you must specify `typename`. This is another example of arcane C++ syntax.

The class also prevents assignment and pass-by-value because of the `mCurElem` data member. To make assignment and pass-by-value work, you would have to implement an assignment operator and copy constructor and make sure `mCurElem` is valid in the destination object. This is omitted in this example.

The implementation of the `RoundRobin` class follows with comments explaining the code. Note the use of `reserve()` in the constructor, and the extensive use of the iterator in `add()`, `remove()`, and `getNext()`. The trickiest aspect is handling `mCurElem` in the `add()` and `remove()` methods.

```

template <typename T> RoundRobin<T>::RoundRobin(size_t numExpected)
{
    // If the client gave a guideline, reserve that much space.
    mElems.reserve(numExpected);

```

```
// Initialize mCurElem even though it isn't used until
// there's at least one element.
mCurElem = end(mElems);
}

template <typename T> RoundRobin<T>::~RoundRobin()
{
    // nothing to do here -- the vector will delete all the elements
}

// Always add the new element at the end
template <typename T> void RoundRobin<T>::add(const T& elem)
{
    // Even though we add the element at the end, the vector could
    // reallocate and invalidate the iterator with the push_back() call.
    // Take advantage of the random access iterator features to save our
    // spot. When getNext() hasn't been called yet, mCurElem is equal
    // to end(mElems) (see constructor), in which case pos is set to -1.
    int pos = (mCurElem == end(mElems)) ? -1 : mCurElem - begin(mElems));
    // Add the element.
    mElems.push_back(elem);
    // Reset our iterator to make sure it is valid.
    // If getNext() hasn't been called yet, reset mCurElem to end(mElems).
    mCurElem = (pos == -1 ? end(mElems) : begin(mElems) + pos);
}

template <typename T> void RoundRobin<T>::remove(const T& elem)
{
    for (auto it = begin(mElems); it != end(mElems); ++it) {
        if (*it == elem) {
            // Removing an element will invalidate our mCurElem iterator if
            // it refers to an element past the point of the removal.
            // Take advantage of the random access features of the iterator
            // to track the position of the current element after removal.
            int newPos;
            // If current iterator is before or at the one we're removing,
            // the new position is the same as before.
            if (mCurElem <= it) {
                newPos = mCurElem - begin(mElems);
            } else {
                // otherwise, it's one less than before
                newPos = mCurElem - begin(mElems) - 1;
            }
            // Erase the element (and ignore the return value).
            mElems.erase(it);
            // Now reset our iterator to make sure it is valid.
            mCurElem = begin(mElems) + newPos;
            return;
        }
    }
}

template <typename T> T& RoundRobin<T>::getNext()
{
    // First, make sure there are any elements.
    if (mElems.empty()) {
```

```

        throw std::out_of_range("No elements in the list");
    }
    // If getNext() hasn't been called yet, mCurElem is equal to end(mElems)
    // (see constructor). In that case wrap to the beginning.
    if (mCurElem == end(mElems)) {
        mCurElem = begin(mElems);
    } else {
        // getNext() has been called before.
        // Increment the iterator modulo the number of elements.
        ++mCurElem;
        if (mCurElem == end(mElems)) {
            mCurElem = begin(mElems);
        }
    }
    // Return a reference to the element.
    return *mCurElem;
}

```

Here's a simple implementation of a scheduler that uses the RoundRobin class template, with comments explaining the code:

```

// Simple Process class.
class Process
{
public:
    // Constructor accepting the name of the process.
    Process(const string& name) : mName(name) {}
    // Implementation of doWorkDuringTimeSlice would let the process
    // perform its work for the duration of a time slice.
    // Actual implementation omitted.
    void doWorkDuringTimeSlice() {
        cout << "Process " << mName
            << " performing work during time slice." << endl;
    }
    // Needed for the RoundRobin::remove method to work.
    bool operator==(const Process& rhs) {
        return mName == rhs.mName;
    }
private:
    string mName;
};

// Simple round-robin based process scheduler.
class Scheduler
{
public:
    // Constructor takes a vector of processes.
    Scheduler(const vector<Process>& processes);
    // Selects the next process using a round-robin scheduling
    // algorithm and allows it to perform some work during
    // this time slice.
    void scheduleTimeSlice();
    // Removes the given process from the list of processes.
    void removeProcess(const Process& process);
private:
    RoundRobin<Process> rr;
}

```

```
};

Scheduler::Scheduler(const vector<Process>& processes)
{
    // Add the processes
    for (auto& process : processes) {
        rr.add(process);
    }
}
void Scheduler::scheduleTimeSlice()
{
    try {
        rr.getNext().doWorkDuringTimeSlice();
    } catch (const out_of_range& {
        cerr << "No more processes to schedule." << endl;
    }
}
void Scheduler::removeProcess(const Process& process)
{
    rr.remove(process);
}
int main()
{
    vector<Process> processes = { Process("1"), Process("2"), Process("3") };
    Scheduler sched(processes);
    for (int i = 0; i < 4; ++i)
        sched.scheduleTimeSlice();
    sched.removeProcess(processes[1]);
    cout << "Removed second process" << endl;
    for (int i = 0; i < 4; ++i)
        sched.scheduleTimeSlice();
    return 0;
}
```

The output should be as follows:

```
Process 1 performing work during time slice.
Process 2 performing work during time slice.
Process 3 performing work during time slice.
Process 1 performing work during time slice.
Removed second process
Process 3 performing work during time slice.
Process 1 performing work during time slice.
Process 3 performing work during time slice.
Process 1 performing work during time slice.
```

The `vector<bool>` Specialization

The standard requires a partial specialization of `vector` for `bool`, with the intention that it optimizes space allocation by “packing” the Boolean values. Recall that a `bool` is either `true` or `false`, and thus could be represented by a single bit, which can take on exactly two values. C++ does not have a native type that stores exactly one bit. Some compilers represent a Boolean value with a type the same size as a `char`. Some other compilers use an `int`. The `vector<bool>` specialization is supposed to store the “array of `bools`” in single bits, thus saving space.

NOTE You can think of the `vector<bool>` as a bit-field instead of a vector. The `bitset` container described later in this chapter provides a more full-featured bit-field implementation than does `vector<bool>`. However, the benefit of `vector<bool>` is that it can change size dynamically.

In a half-hearted attempt to provide some bit-field routines for the `vector<bool>`, there is one additional method: `flip()`. This method can be called on either the container, in which case it complements all the elements in the container; or on a single reference returned from `operator[]` or a similar method, in which case it complements that single element.

At this point, you should be wondering how you can call a method on a reference to `bool`. The answer is that you can't. The `vector<bool>` specialization actually defines a class called `reference` that serves as a proxy for the underlying `bool` (or bit). When you call `operator[]`, `at()`, or a similar method, the `vector<bool>` returns a `reference` object, which is a proxy for the real `bool`.

WARNING The fact that references returned from `vector<bool>` are really proxies means that you can't take their addresses to obtain pointers to the actual elements in the container.

In practice, the little amount of space saved by packing `bools` hardly seems worth the extra effort. Even worse, accessing and modifying elements in a `vector<bool>` is much slower than, for example, in a `vector<int>`. Many C++ experts recommend avoiding `vector<bool>` in favor of the `bitset`, or `vector<int>` if you need a dynamically sized bit field.

deque

`deque` (abbreviation for double-ended queue) is almost identical to `vector`, but is used far less frequently. It is defined in the `<deque>` header file. The principle differences are as follows:

- Elements are not stored contiguously in memory.
- A `deque` supports true constant-time insertion and removal of elements at both the front and the back (a `vector` supports amortized constant time at just the back).
- A `deque` provides `push_front()`, `pop_front()`, and `emplace_front()`, which the `vector` omits.
- A `deque` does not expose its memory management scheme via `reserve()` or `capacity()`.

`deques` are rarely used, as opposed to `vectors` and `lists`, so they are not further discussed. Consult a Standard Library Reference for a detailed list of all supported methods.

list

The STL `list` class, defined in the `<list>` header file, is a standard doubly linked list. It supports constant-time insertion and deletion of elements at any point in the list, but provides slow (linear) time access to individual elements. In fact, the `list` does not even provide random-access operations like `operator[]`. Only through iterators can you access individual elements.

Most of the `list` operations are identical to those of `vector`, including the constructors, destructor, copying operations, assignment operations, and comparison operations. This section focuses on those methods that differ from those of `vector`.

Accessing Elements

The only methods provided by a `list` to access elements are `front()` and `back()`, both of which run in constant time. These methods return a reference to the first and last element in a `list`. All other element access must be performed through iterators.

A `list` supports `begin()`, returning an iterator referring to the first element in the `list`, and `end()`, returning an iterator referring to one past the last element in the `list`. It also supports `cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, and `crend()`.

WARNING *Lists do not provide random access to elements.*

Iterators

A `list` iterator is bidirectional, not random access like a `vector` iterator. That means that you cannot add and subtract `list` iterators from each other, or perform other pointer arithmetic on them. For example, if `p` is a `list` iterator, you can traverse through the elements of the `list` by doing `++p` or `--p`, but you cannot use the addition or subtraction operator; `p+n` or `p-n` does not work.

Adding and Removing Elements

A `list` supports the same add element and remove element methods as a `vector`, including `push_back()`, `pop_back()`, `emplace()`, `emplace_back()`, the five forms of `insert()`, the two forms of `erase()`, and `clear()`. Like a `deque`, it also provides `push_front()`, `emplace_front()`, and `pop_front()`. The amazing thing about a `list` is that all these methods (except for `clear()`) run in constant time, once you've found the correct position. Thus, a `list` is appropriate for applications that perform many insertions and deletions from the data structure, but do not need quick index-based element access.

list Size

Like `deques`, and unlike `vectors`, `lists` do not expose their underlying memory model. Consequently, they support `size()`, `empty()` and `resize()`, but not `reserve()` or `capacity()`. Note that the `size()` method on a `list` has constant complexity, which is not the case for the `size()` method on a `forward_list`.

Special list Operations

A `list` provides several special operations that exploit its quick element insertion and deletion. This section provides an overview and examples. Consult a Standard Library Reference — for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/> — for a thorough reference of all the methods.

Splicing

The linked-list characteristics of a `list` allow it to *splice*, or insert, an entire `list` at any position in another `list` in constant time. The simplest version of this method works as follows:

```
// Store the a words in the main dictionary.
list<string> dictionary{ "aardvark", "ambulance" };
// Store the b words.
list<string> bWords{ "bathos", "balderdash" };
// Add the c words to the main dictionary
dictionary.push_back("canticle");
dictionary.push_back("consumerism");
// splice the b words into the main dictionary.
if (bWords.size() > 0) {
    // Get an iterator to the last b word.
    auto iterLastB = --(cend(bWords));
    // Iterate up to the spot where we want to insert bs.
    auto it = cbegin(dictionary);
    for (; it != cend(dictionary); ++it) {
        if (*it > *iterLastB)
            break;
    }
    // Add in the bwords. This action removes the elements from bWords.
    dictionary.splice(it, bWords);
}
// print out the dictionary
for (const auto& word : dictionary) {
    cout << word << endl;
}
```

The result from running this program looks like this:

```
aardvark
ambulance
bathos
balderdash
canticle
consumerism
```

There are also two other forms of `splice()`: one that inserts a single element from another `list` and one that inserts a range from another `list`. Additionally, all forms of `splice()` are available with either a normal reference or a `rvalue` reference to the source `list`.

WARNING *Splicing is destructive to the list passed as an argument: It removes the spliced elements from one list in order to insert them into the other.*

More Efficient Versions of Algorithms

In addition to `splice()`, a `list` provides special implementations of several of the generic STL algorithms. The generic forms are covered in Chapter 17. Here, only the specific versions provided by `list` are discussed.

NOTE When you have a choice, use the list specific methods rather than the generic STL algorithms because the former are more efficient. Sometimes you don't have a choice and you must use the list specific methods; for example, `std::sort()` requires RandomAccessIterators, which a list does not provide.

The following table summarizes the algorithms for which `list` provides special implementations as methods. See Chapter 17 for more details on the algorithms.

METHOD	DESCRIPTION
<code>remove()</code>	Removes certain elements from a list.
<code>remove_if()</code>	
<code>unique()</code>	Removes duplicate consecutive elements from a list, based on <code>operator==</code> or a user-supplied binary predicate.
<code>merge()</code>	Merges two lists. Both lists must be sorted to start, according to <code>operator<</code> or a user-defined comparator. Like <code>splice()</code> , <code>merge()</code> is destructive to the list passed as an argument.
<code>sort()</code>	Performs a stable sort on elements in a list.
<code>reverse()</code>	Reverses the order of the elements in a list.

list Example: Determining Enrollment

Suppose that you are writing a computer registration system for a university. One feature you might provide is the ability to generate a complete list of enrolled students in the university from lists of the students in each class. For the sake of this example, assume that you must write only a single function that takes a vector of lists of student names (as `strings`), plus a list of students that have been dropped from their courses because they failed to pay tuition. This method should generate a complete list of all the students in all the courses, without any duplicates, and without those students who have been dropped. Note that students might be in more than one course.

Here is the code for this method, with comments explaining the code. With the power of STL lists, the method is practically shorter than its written description! Note that the STL allows you to “nest” containers: in this case, you can use a vector of lists.

```

// courseStudents is a vector of lists, one for each course. The lists
// contain the students enrolled in those courses. They are not sorted.
//
// droppedStudents is a list of students who failed to pay their
// tuition and so were dropped from their courses.
//
// The function returns a list of every enrolled (non-dropped) student in
// all the courses.
list<string> getTotalEnrollment(const vector<list<string>>& courseStudents,
                                const list<string>& droppedStudents)

```

```

{
    list<string> allStudents;
    // Concatenate all the course lists onto the master list
    for (auto& lst : courseStudents) {
        allStudents.insert(cend(allStudents), cbegin(lst), cend(lst));
    }
    // Sort the master list
    allStudents.sort();
    // Remove duplicate student names (those who are in multiple courses).
    allStudents.unique();
    // Remove students who are on the dropped list.
    // Iterate through the drop list, calling remove on the
    // master list for each student in the dropped list.
    for (auto& str : droppedStudents) {
        allStudents.remove(str);
    }
    // done!
    return allStudents;
}

```

forward_list

A `forward_list`, defined in the `<forward_list>` header file, is similar to a `list` except that it is a singly linked list while a `list` is a doubly linked list. This means that `forward_list` supports only forward iteration and, because of this, ranges need to be specified differently compared to a `list`. If you want to modify any list, you need access to the element before the first element of interest. Since a `forward_list` does not have an iterator that supports going backward, there is no easy way to get to the preceding element. For this reason, ranges that will be modified; for example, ranges supplied to `erase()` and `splice()`, must be open at the beginning. The `begin()` function seen earlier returns an iterator to the first element and thus can be used only to construct a range that is closed at the beginning. The `forward_list` class therefore defines a `before_begin()` method, which returns an iterator that points to an imaginary element before the beginning of the list. You cannot dereference this iterator as it points to invalid data. However, incrementing this iterator by one will make it the same as the iterator returned by `begin()`; so it can be used to make a range that is open at the beginning. The following table sums up the differences between a `list` and a `forward_list`:

OPERATION	LIST	FORWARD_LIST
<code>assign()</code>	x	x
<code>back()</code>	x	
<code>before_begin()</code>		x
<code>begin()</code>	x	x
<code>cbefore_begin()</code>		x
<code>cbegin()</code>	x	x
<code>cend()</code>	x	x

continues

(continued)

OPERATION	LIST	FORWARD_LIST
clear()	x	x
crbegin()	x	
crend()	x	
emplace()	x	
emplace_after()		x
emplace_back()	x	
emplace_front()	x	x
empty()	x	x
end()	x	x
erase()	x	
erase_after()		x
front()	x	x
insert()	x	
insert_after()		x
iterator / const_iterator	x	x
max_size()	x	x
merge()	x	x
pop_back()	x	
pop_front()	x	x
push_back()	x	
push_front()	x	x
rbegin()	x	
remove()	x	x
remove_if()	x	x
rend()	x	
resize()	x	x
reverse()	x	x

reverse_iterator / const_reverse_iterator	x	
size()	x	
sort()	x	x
splice()	x	
splice_after()		x
swap()	x	x
unique()	x	x

Constructors and assignment operators are similar between a `list` and a `forward_list`. The C++ standard states that `forward_lists` should try to use minimal space. That's the reason why there is no `size()` method, because by not providing it, there is no need to store the size of the list. The following example demonstrates the use of `forward_lists`:

```
// Create 3 forward lists and use an initializer_list
// to initialize their elements (uniform initialization).
forward_list<int> lst1 = { 5, 6 };
forward_list<int> lst2 = { 1, 2, 3, 4 };
forward_list<int> lst3 = { 7, 8, 9 };
// Insert lst2 at the front of lst1 using splice.
lst1.splice_after(lst1.before_begin(), lst2);
// Add number 0 at the beginning of the lst1.
lst1.push_front(0);
// Insert lst3 at the end of lst1.
// For this, we first need an iterator to the last element.
auto iter = lst1.before_begin();
auto iterTemp = iter;
while (++iterTemp != end(lst1)) {
    ++iter;
}
lst1.insert_after(iter, cbegin(lst3), cend(lst3));
// Output the contents of lst1.
for (auto& i : lst1) {
    cout << i << ' ';
}
```

To insert `lst3`, we need an iterator to the last element in the list. However, since this is a `forward_list`, we cannot use `--end(lst1)`, so we need to iterate over the list from the beginning and stop at the last element. The output of this example is as follows:

```
0 1 2 3 4 5 6 7 8 9
```

array

An array, defined in the `<array>` header file, is similar to a `vector` except that it is of a fixed size; it cannot grow or shrink in size. The purpose of this is to allow an array to be allocated on the stack, rather than always demanding heap access as `vector` does. Just like `vectors`, arrays support random-access iterators, and elements are stored in contiguous memory. It has support for `front()`,

`back()`, `at()`, and `operator[]`. It also supports a `fill()` method to fill the array with a specific element. Because it is fixed in size, it does not support `push_back()`, `pop_back()`, `insert()`, `erase()`, `clear()`, `resize()`, `reserve()`, and `capacity()`. A disadvantage compared to a vector is that the `swap()` method of an array runs in linear time while it has constant complexity for a vector. Arrays can also not be moved in constant time, while vectors can. An array has a `size()` method, which is a clear advantage over C-style arrays. The following example demonstrates how to use the `array` class. Note that the `array` declaration requires two template parameters; the first specifies the type of the elements, and the second specifies the fixed number of elements in the array.

```
// Create array of 3 integers and initialize them
// with the given initializer_list using uniform initialization.
array<int, 3> arr = { 9, 8, 7 };
// Output the size of the array.
cout << "Array size = " << arr.size() << endl;
// Output the contents using the range-based for loop.
for (const auto& i : arr) {
    cout << i << endl;
}

cout << "Performing arr.fill(3)..." << endl;
// Use the fill method to change the contents of the array.
arr.fill(3);
// Output the contents of the array using iterators.
for (auto iter = cbegin(arr); iter != cend(arr); ++iter) {
    cout << *iter << endl;
}
```

The output of the preceding code is as follows:

```
Array size = 3
9
8
7
Performing arr.fill(3)...
3
3
3
```

CONTAINER ADAPTERS

In addition to the standard sequential containers, the STL provides three container adapters: `queue`, `priority_queue`, and `stack`. Each of these adapters is a wrapper around one of the sequential containers. They allow you to swap the underlying container without having to change the rest of the code. The intent of the adapters is to simplify the interface and to provide only those features that are appropriate for the `stack` or `queue` abstraction. For example, the adapters don't provide iterators or the capability to insert or erase multiple elements simultaneously.

queue

The `queue` container adapter, defined in the header file `<queue>`, provides standard “first-in, first-out” (FIFO) semantics. As usual, it's written as a class template, which looks like this:

```
template <class T, class Container = deque<T> > class queue;
```

The `T` template parameter specifies the type that you intend to store in the queue. The second template parameter allows you to stipulate the underlying container that the queue adapts. However, the queue requires the sequential container to support both `push_back()` and `pop_front()`, so you have only two built-in choices: `deque` and `list`. For most purposes, you can just stick with the default `deque`.

queue Operations

The queue interface is extremely simple: there are only eight methods plus the constructor and the normal comparison operators. The `push()` and `emplace()` methods add a new element to the tail of the queue, and `pop()` removes the element at the head of the queue. You can retrieve references to, without removing, the first and last elements with `front()` and `back()`, respectively. As usual, when called on `const` objects, `front()` and `back()` return `const` references; and when called on non-`const` objects they return non-`const` (read/write) references.

WARNING `pop()` does not return the element popped. If you want to retain a copy, you must first retrieve it with `front()`.

The queue also supports `size()`, `empty()`, and `swap()`.

queue Example: A Network Packet Buffer

When two computers communicate over a network, they send information to each other divided up into discrete chunks called *packets*. The networking layer of the computer's operating system must pick up the packets and store them as they arrive. However, the computer might not have enough bandwidth to process all of them at once. Thus, the networking layer usually *buffers*, or stores, the packets until the higher layers have a chance to attend to them. The packets should be processed in the order they arrive, so this problem is perfect for a queue structure. The following is a small `PacketBuffer` class, with comments explaining the code, which stores incoming packets in a queue until they are processed. It's a template so that different layers of the networking layer can use it for different kinds of packets, such as IP packets or TCP packets. It allows the client to specify a maximum size because operating systems usually limit the number of packets that can be stored, so as not to use too much memory. When the buffer is full, subsequently arriving packets are ignored.

```
template <typename T>
class PacketBuffer
{
public:
    // If maxSize is 0, the size is unlimited, because creating
    // a buffer of size 0 makes little sense. Otherwise only
    // maxSize packets are allowed in the buffer at any one time.
    PacketBuffer(size_t maxSize = 0);
    // Stores a packet in the buffer.
    // Returns false if the packet has been discarded because
    // there is no more space in the buffer, true otherwise.
    bool bufferPacket(const T& packet);
    // Returns the next packet. Throws out_of_range
    // if the buffer is empty.
```

```
        T getNextPacket();
private:
    std::queue<T> mPackets;
    size_t mMaxSize;
};

template <typename T> PacketBuffer<T>::PacketBuffer(size_t maxSize/* = 0 */)
    : mMaxSize(maxSize)
{
}

template <typename T> bool PacketBuffer<T>::bufferPacket(const T& packet)
{
    if (mMaxSize > 0 && mPackets.size() == mMaxSize) {
        // No more space. Drop the packet.
        return false;
    }
    mPackets.push(packet);
    return true;
}

template <typename T> T PacketBuffer<T>::getNextPacket()
{
    if (mPackets.empty()) {
        throw std::out_of_range("Buffer is empty");
    }
    // retrieve the head element
    T temp = mPackets.front();
    // pop the head element
    mPackets.pop();
    // return the head element
    return temp;
}
```

A practical application of this class would require multiple threads. C++ provides synchronization classes to allow thread-safe access to shared objects. Without explicit synchronization, no STL class can be used safely from multiple threads. Synchronization is discussed in Chapter 23. The focus in this example is on the queue class, so here is a single-threaded example of using the `PacketBuffer`:

```
class IPPacket
{
public:
    IPPacket(int id) : mID(id) {}
    int getID() const { return mID; }
private:
    int mID;
};

int main()
{
    PacketBuffer<IPPacket> ipPackets(3);
    // Add 4 packets
    for (int i = 1; i <= 4; ++i) {
        if (!ipPackets.bufferPacket(IPPacket(i))) {
            cout << "Packet " << i << " dropped (queue is full)." << endl;
```

```

        }
    }
    while (true) {
        try {
            IPPacket packet = ipPackets.getNextPacket();
            cout << "Processing packet " << packet.getID() << endl;
        } catch (const out_of_range& ) {
            cout << "Queue is empty." << endl;
            break;
        }
    }
    return 0;
}

```

The output of this program is as follows:

```

Packet 4 dropped (queue is full).
Processing packet 1
Processing packet 2
Processing packet 3
Queue is empty.

```

priority_queue

A *priority queue* is a queue that keeps its elements in sorted order. Instead of a strict FIFO ordering, the element at the head of the queue at any given time is the one with the highest priority. This element could be the oldest on the queue or the most recent. If two elements have equal priority, their relative order in the queue is undefined.

The `priority_queue` container adapter is also defined in `<queue>`. Its template definition looks something like this (slightly simplified):

```

template <class T, class Container = vector<T>,
          class Compare = less<T> >;

```

It's not as complicated as it looks. You've seen the first two parameters before: `T` is the element type stored in the `priority_queue` and `Container` is the underlying container on which the `priority_queue` is adapted. The `priority_queue` uses `vector` as the default, but `deque` works as well. `list` does not work because the `priority_queue` requires random access to its elements. The third parameter, `Compare`, is trickier. As you'll learn more about in Chapter 17, `less` is a class template that supports comparison of two objects of type `T` with `operator<`. What this means for you is that the priority of elements in the `queue` is determined according to `operator<`. You can customize the comparison used, but that's a topic for Chapter 17. For now, just make sure that you define `operator<` appropriately for the types stored in the `priority_queue`.

NOTE *The head element of the `priority_queue` is the one with the “highest” priority, by default, determined according to `operator<` such that elements that are “less” than other elements have lower priority.*

priority_queue Operations

The `priority_queue` provides even fewer operations than does the `queue`. The `push()` and `emplace()` methods allow you to insert elements, `pop()` allows you to remove elements, and `top()` returns a `const` reference to the head element.

WARNING `top()` returns a `const` reference even when called on a non-`const` object, because modifying the element might change its order, which is not allowed. The `priority_queue` provides no mechanism to obtain the tail element.

WARNING `pop()` does not return the element popped. If you want to retain a copy, you must first retrieve it with `top()`.

Like the `queue`, the `priority_queue` supports `size()`, `empty()`, and `swap()`. However, it does not provide any comparison operators.

priority_queue Example: An Error Correlator

Single failures on a system can often cause multiple errors to be generated from different components. A good error-handling system uses *error correlation* to process the most important errors first. You can use a `priority_queue` to write a very simple error correlator. Assume all error events encode their own priority. The error correlator simply sorts error events according to their priority, so that the highest-priority errors are always processed first. Here is the class definition:

```
// Sample Error class with just a priority and a string error description.
class Error
{
public:
    Error(int priority, const std::string& errMsg)
        : mPriority(priority), mError(errMsg) {}
    int getPriority() const { return mPriority; }
    const std::string& getErrorString() const { return mError; }
    friend bool operator<(const Error& lhs, const Error& rhs);
    friend std::ostream& operator<<(std::ostream& os, const Error& err);
private:
    int mPriority;
    std::string mError;
};

// Simple ErrorCorrelator class that returns highest priority errors first.
class ErrorCorrelator
{
public:
    // Add an error to be correlated.
    void addError(const Error& error);
    // Retrieve the next error to be processed.
    Error getError();
private:
    std::priority_queue<Error> mErrors;
```

};

Here are the definitions of the functions and methods:

```

bool operator<(const Error& lhs, const Error& rhs)
{
    return (lhs.mPriority < rhs.mPriority);
}
ostream& operator<<(ostream& os, const Error& err)
{
    os << err.mError << " (priority " << err.mPriority << ")";
    return os;
}
void ErrorCorrelator::addError(const Error& error)
{
    mErrors.push(error);
}
Error ErrorCorrelator::getError()
{
    // If there are no more errors, throw an exception.
    if (mErrors.empty())
        throw out_of_range("No more errors.");
    }
    // Save the top element.
    Error top = mErrors.top();
    // Remove the top element.
    mErrors.pop();
    // Return the saved element.
    return top;
}

```

Here is a simple unit test showing how to use the `ErrorCorrelator`. Realistic use would require multiple threads so that one thread adds errors, while another processes them. As mentioned earlier with the `queue` example, this requires explicit synchronization, discussed in Chapter 23.

```

ErrorCorrelator ec;
ec.addError(Error(3, "Unable to read file"));
ec.addError(Error(1, "Incorrect entry from user"));
ec.addError(Error(10, "Unable to allocate memory!"));
while (true) {
    try {
        Error e = ec.getError();
        cout << e << endl;
    } catch (const out_of_range& e) {
        cout << "Finished processing errors" << endl;
        break;
    }
}

```

The output of this program is as follows:

```

Unable to allocate memory! (priority 10)
Unable to read file (priority 3)
Incorrect entry from user (priority 1)
Finished processing errors

```

stack

A stack is almost identical to a queue, except that it provides *first-in, last-out* (FILO) semantics, also known as *last-in, first-out* (LIFO), instead of FIFO. It is defined in the `<stack>` header file. The template definition looks like this:

```
template <class T, class Container = deque<T> > class stack;
```

You can use `vector`, `list`, or `deque` as the underlying container for the stack.

stack Operations

Like the queue, the stack provides `push()`, `emplace()`, and `pop()`. The difference is that `push()` adds a new element to the top of the stack, “pushing down” all elements inserted earlier, and `pop()` removes the element from the top of the stack, which is the most recently inserted element. The `top()` method returns a `const` reference to the top element if called on a `const` object and a non-`const` reference if called on a non-`const` object.

WARNING `pop()` does not return the element popped. If you want to retain a copy, you must first retrieve it with `top()`.

The stack supports `empty()`, `size()`, `swap()`, and the standard comparison operators.

stack Example: Revised Error Correlator

You can rewrite the previous `ErrorCorrelator` class so that it gives out the most recent error instead of the one with the highest priority. The only change required is to change `mErrors` from a `priority_queue` to a `stack`. With this change, the errors will be distributed in LIFO order instead of priority order. Nothing in the method definitions needs to change because the `push()`, `pop()`, `top()`, and `empty()` methods exist on both the `priority_queue` and `stack`.

ASSOCIATIVE CONTAINERS

Unlike the sequential containers, the associative containers do not store elements in a linear configuration. Instead, they provide a mapping of keys to values. They generally offer insertion, deletion, and lookup times that are equivalent to each other.

There are four ordered associative containers provided by the STL: `map`, `multimap`, `set`, and `multiset`. Each of these containers stores its elements in a *sorted*, tree-like, data structure. There are also four unordered associative containers: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. These are discussed later in this chapter.

The pair Utility Class

Before learning about the associative containers, you must become familiar with the `pair` class, which is defined in the `<utility>` header file. `pair` is a class template that groups together two values of possibly different types. The values are accessible through the `first` and `second` public

data members. `operator==` and `operator<` are defined for pairs to compare both the first and second elements. Here are some examples:

```
// Two-argument constructor and default constructor
pair<string, int> myPair("hello", 5);
pair<string, int> myOtherPair;
// Can assign directly to first and second
myOtherPair.first = "hello";
myOtherPair.second = 6;
// Copy constructor
pair<string, int> myThirdPair(myOtherPair);
// operator<
if (myPair < myOtherPair) {
    cout << "myPair is less than myOtherPair" << endl;
} else {
    cout << "myPair is greater than or equal to myOtherPair" << endl;
}
// operator==
if (myOtherPair == myThirdPair) {
    cout << "myOtherPair is equal to myThirdPair" << endl;
} else {
    cout << "myOtherPair is not equal to myThirdPair" << endl;
}
```

The output is as follows:

```
myPair is less than myOtherPair
myOtherPair is equal to myThirdPair
```

The library also provides a utility function template, `make_pair()`, that constructs a pair from two values. For example:

```
pair<int, int> aPair = make_pair(5, 10);
```

Of course, in this case you could have just used the two-argument constructor. However, `make_pair()` is more useful when you want to pass a pair to a function, or assign it to a pre-existing variable. Unlike class templates, function templates can infer types from parameters, so you can use `make_pair()` to construct a pair without explicitly specifying the types. You can also use `make_pair()` in combination with the `auto` keyword as follows:

```
auto aSecondPair = make_pair(5, 10);
```

WARNING *Using plain old pointer types in pairs is risky because the pair copy constructor and assignment operator perform only shallow copies and assignments of pointer types. However, you can safely store smart pointers like shared_ptr in a pair.*

map

A `map`, defined in the `<map>` header file, stores key/value pairs instead of just single values. Insertion, lookup, and deletion are all based on the key; the value is just “along for the ride.” The term “map” comes from the conceptual understanding that the container “maps” keys to values.

A `map` keeps elements in sorted order, based on the keys, so that insertion, deletion, and lookup all take logarithmic time. Because of the order, when you enumerate the elements, they come out in the ordering imposed by the type's `operator<` or a user-defined comparator. It is usually implemented as some form of balanced tree, such as a red-black tree. However, the tree structure is not exposed to the client.

You should use a `map` whenever you need to store and retrieve elements based on a “key” and you would like to have them in a certain order.

Constructing maps

The `map` template takes four types: the key type, the value type, the comparison type, and the allocator type. As always, the allocator is ignored in this chapter. The comparison type is similar to the comparison type for `priority_queue` described earlier. It allows you to specify a different comparison class than the default. In this chapter, only the default `less` comparison is used. When using the default, make sure that your keys all respond to `operator<` appropriately. If you're interested in further detail, Chapter 17 explains how to write your own comparison classes.

If you ignore the comparison and allocator parameters, constructing a `map` is just like constructing a `vector` or a `list`, except that you specify the key and value types separately in the template. For example, the following code constructs a `map` that uses `ints` as the key and stores objects of the `Data` class:

```
class Data
{
public:
    explicit Data(int val = 0) { mVal = val; }
    int getVal() const { return mVal; }
    void setVal(int val) { mVal = val; }
private:
    int mVal;
};

...
map<int, Data> dataMap;
```

`maps` also support uniform initialization:

```
map<string, int> m = {
    { "Marc G.", 123 },
    { "Warren B.", 456 },
    { "Peter V.W.", 789 }
};
```

Inserting Elements

Inserting an element into sequential containers such as `vector` and `list` always requires you to specify the position at which the element is to be added. A `map`, along with the other associative containers, is different. The `map` internal implementation determines the position in which to store the new element; you need only to supply the key and the value.

NOTE *map and the other associative containers do provide a version of insert() that takes an iterator position. However, that position is only a “hint” to the container as to the correct position. The container is not required to insert the element at that position.*

When inserting elements, it is important to keep in mind that `maps` require unique keys: every element in the `map` must have a different key. If you want to support multiple elements with the same key, you have two options. You can either use a `map` and store another container such as a `vector` or an `array` as the element for a key, or you can use `multimaps`, described later.

There are two ways to insert an element into the `map`: one clumsy and one not so clumsy.

The `insert()` Method

The clumsy mechanism to add an element to a `map` is the `insert()` method, but it has the advantage of allowing you to detect if the key already exists. One problem is that you must specify the key/value pair as a `pair` object or as an `initializer_list`. The second problem is that the return value from the basic form of `insert()` is a pair of an iterator and a `bool`. The reason for the complicated return value is that `insert()` does not overwrite an element value if one already exists with the specified key. The `bool` element of the returned `pair` specifies whether the `insert()` actually inserted the new key/value pair or not. The iterator refers to the element in the `map` with the specified key (with a new or old value, depending on whether the `insert` succeeded or failed). `map` iterators are discussed in more detail in the next section. Continuing the `map` example from the previous section, you can use `insert()` as follows:

```
map<int, Data> dataMap;
auto ret = dataMap.insert({ 1, Data(4) }); // Using an initializer_list
if (ret.second) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}
ret = dataMap.insert(make_pair(1, Data(6))); // Using a pair object
if (ret.second) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}
```

The output of the program is as follows:

```
Insert succeeded!
Insert failed!
```

Without the `auto` keyword, you would have to declare the correct type for `ret` yourself as follows:

```
pair<map<int, Data>::iterator, bool> ret;
```

The type of `ret` is a `pair`. The first element of the `pair` is a `map` iterator for a `map` with keys of type `int` and values of type `Data`. The second element of the `pair` is a Boolean value.

operator[]

The less clumsy way to insert an element into a `map` is through the overloaded `operator[]`. The difference is mainly in the syntax: you specify the key and value separately. Additionally, `operator[]` always succeeds. If no element value with the given key exists, it creates a new element with that key and value. If an element with the key exists already, `operator[]` replaces the element value with the newly specified value. Here is the previous example using `operator[]` instead of `insert()`:

```
map<int, Data> dataMap;
dataMap[1] = Data(4);
dataMap[1] = Data(6); // Replaces the element with key 1
```

There is, however, one major caveat to `operator[]`: it always constructs a new value object, even if it doesn't need to use it. Thus, it requires a default constructor for your element values, and can be less efficient than `insert()`.

The fact that `operator[]` creates a new element in a `map` if the requested element does not already exist means that this operator is not marked as `const`. This sounds obvious, but might sometimes look counter intuitive. For example, suppose you have the following function:

```
void func(const map<int, int>& m)
{
    cout << m[1] << endl; // Error
}
```

This will fail to compile, even though you appear to be just reading the value `m[1]`. It fails because the variable `m` is a `const` reference to a `map`, and `operator[]` is not marked as `const`. Instead, you should use the `find()` method described in the “Looking Up Elements” section.

map Iterators

`map` iterators work similarly to the iterators on the sequential containers. The major difference is that the iterators refer to key/value pairs instead of just the values. In order to access the value, you must retrieve the second field of the pair object. Here is how you can iterate through the `map` from the previous example:

```
for (auto iter = cbegin(dataMap); iter != cend(dataMap); ++iter) {
    cout << iter->second.getVal() << endl;
}
```

Take another look at the expression used to access the value:

```
iter->second.getVal()
```

`iter` refers to a key/value pair, so you can use the `->` operator to access the second field of that pair, which is a `Data` object. You can then call the `getVal()` method on that `Data` object.

Note that the following code is functionally equivalent:

```
(*iter).second.getVal()
```

Using the range-based `for` loop, the loop can be written even more elegantly as follows:

```
for (const auto& p : dataMap) {
    cout << p.second.getVal() << endl;
}
```

WARNING You can modify element values through non-const iterators, but the compiler will generate an error if you try to modify the key of an element, even through a non-const iterator, because it would destroy the sorted order of the elements in the map.

map iterators are bidirectional, meaning you can traverse them in both directions.

Looking Up Elements

A map provides logarithmic lookup of elements based on a supplied key. If you already know that an element with a given key is in a map, the simplest way to look it up is through operator[] as long as you call it on a non-const map or a non-const reference to a map. The nice thing about operator[] is that it returns a reference to the element that you can use and modify directly, without worrying about pulling the value out of a pair object. Here is an extension to the preceding example to call the setVal() method on the Data object value at key 1:

```
map<int, Data> dataMap;
dataMap[1] = Data(4);
dataMap[1] = Data(6);
dataMap[1].setVal(100);
```

However, if you don't know whether the element exists, you may not want to use operator[], because it will insert a new element with that key if it doesn't find one already. As an alternative, map provides a find() method that returns an iterator referring to the element with the specified key, if it exists, or the end() iterator if it's not in the map. Here is an example using find() to perform the same modification to the Data object with key 1:

```
auto it = dataMap.find(1);
if (it != end(dataMap)) {
    it->second.setVal(100);
}
```

As you can see, using find() is a bit clumsier, but it's sometimes necessary.

If you only want to know whether or not an element with a certain key is in a map, you can use the count() method. It returns the number of elements in a map with a given key. For maps, the result will always be 0 or 1 because there can be no elements with duplicate keys.

Removing Elements

A map allows you to remove an element at a specific iterator position or to remove all elements in a given iterator range, in amortized constant and logarithmic time, respectively. From the client perspective, these two erase() methods are equivalent to those in the sequential containers. A great feature of a map, however, is that it also provides a version of erase() to remove an element matching a key. Here is an example:

```
map<int, Data> dataMap;
dataMap[1] = Data(4);
cout << "There are " << dataMap.count(1) << " elements with key 1" << endl;
dataMap.erase(1);
cout << "There are " << dataMap.count(1) << " elements with key 1" << endl;
```

The output is as follows:

```
There are 1 elements with key 1
There are 0 elements with key 1
```

map Example: Bank Account

You can implement a simple bank account database using a `map`. A common pattern is for the key to be one field of a `class` or `struct` that is stored in a `map`. In this case, the key is the account number. Here are simple `BankAccount` and `BankDB` classes:

```
class BankAccount
{
public:
    BankAccount(int acctNum, const std::string& name)
        : mAcctNum(acctNum), mClientName(name) {}
    void setAcctNum(int acctNum) { mAcctNum = acctNum; }
    int getAcctNum() const { return mAcctNum; }
    void setClientName(const std::string& name) { mClientName = name; }
    const std::string& getClientName() const { return mClientName; }
private:
    int mAcctNum;
    std::string mClientName;
};

class BankDB
{
public:
    // Adds acct to the bank database. If an account exists already
    // with that number, the new account is not added. Returns true
    // if the account is added, false if it's not.
    bool addAccount(const BankAccount& acct);
    // Removes the account acctNum from the database.
    void deleteAccount(int acctNum);
    // Returns a reference to the account represented
    // by its number or the client name.
    // Throws out_of_range if the account is not found.
    BankAccount& findAccount(int acctNum);
    BankAccount& findAccount(const std::string& name);
    // Adds all the accounts from db to this database.
    // Deletes all the accounts from db.
    void mergeDatabase(BankDB& db);
private:
    std::map<int, BankAccount> mAccounts;
};
```

Here are the implementations of the `BankDB` methods, with comments explaining the code:

```
bool BankDB::addAccount(const BankAccount& acct)
{
    // Do the actual insert, using the account number as the key
    auto res = mAccounts.insert(make_pair(acct.getAcctNum(), acct));
    // Return the bool field of the pair specifying success or failure
    return res.second;
}
```

```

void BankDB::deleteAccount(int acctNum)
{
    mAccounts.erase(acctNum);
}
BankAccount& BankDB::findAccount(int acctNum)
{
    // Finding an element via its key can be done with find()
    auto it = mAccounts.find(acctNum);
    if (it == end(mAccounts)) {
        throw out_of_range("No account with that number.");
    }
    // Remember that iterators into maps refer to pairs of key/value
    return it->second;
}
BankAccount& BankDB::findAccount(const string& name)
{
    // Finding an element by a non-key attribute requires a linear
    // search through the elements.
    for (auto& p : mAccounts) {
        if (p.second.getClientName() == name) {
            // found it!
            return p.second;
        }
    }
    throw out_of_range("No account with that name.");
}
void BankDB::mergeDatabase(BankDB& db)
{
    // Just insert copies of all the accounts in the old db
    // to the new database.
    mAccounts.insert(begin(db.mAccounts), end(db.mAccounts));
    // Now delete all the accounts in the old database.
    db.mAccounts.clear();
}

```

You can test the BankDB class with the following code:

```

BankDB db;
db.addAccount(BankAccount(100, "Nicholas Solter"));
db.addAccount(BankAccount(200, "Scott Kleper"));
try {
    auto& acct = db.findAccount(100);
    cout << "Found account 100" << endl;
    acct.setClientName("Nicholas A Solter");
    auto& acct2 = db.findAccount("Scott Kleper");
    cout << "Found account of Scott Kelper" << endl;
    auto& acct3 = db.findAccount(1000);
} catch (const out_of_range& ) {
    cout << "Unable to find account" << endl;
}

```

The output is as follows:

```

Found account 100
Found account of Scott Kelper
Unable to find account

```

multimap

A `multimap` is a `map` that allows multiple elements with the same key. Like `maps`, `multimaps` support uniform initialization. The interface is almost identical to the `map` interface, with the following changes:

- `multimaps` do not provide `operator[]`. The semantics of this operator do not make sense if there can be multiple elements with a single key.
- Inserts on `multimaps` always succeed. Thus, the `multimap insert()` method that adds a single element returns only an iterator.

NOTE *multimaps allow you to insert identical key/value pairs. If you want to avoid this redundancy, you must check explicitly before inserting a new element.*

The trickiest aspect of `multimaps` is looking up elements. You can't use `operator[]`, because it is not provided. `find()` isn't very useful because it returns an iterator referring to any one of the elements with a given key (not necessarily the first element with that key).

However, `multimaps` store all elements with the same key together and provide methods to obtain iterators for this subrange of elements with the same key in the container. The `lower_bound()` and `upper_bound()` methods each return a single iterator referring to the first and one-past-the-last elements matching a given key. If there are no elements matching that key, the iterators returned by `lower_bound()` and `upper_bound()` will be equal to each other.

If you need to obtain both iterators bounding the elements with a given key, it's more efficient to use `equal_range()` instead of calling `lower_bound()` followed by calling `upper_bound()`. `equal_range()` returns a pair of the two iterators that would be returned by `lower_bound()` and `upper_bound()`.

NOTE *The `lower_bound()`, `upper_bound()`, and `equal_range()` methods exist for `maps` as well, but their usefulness is limited because a `map` cannot have multiple elements with the same key.*

multimap Example: Buddy Lists

Most of the numerous online chat programs allow users to have a “buddy list” or list of friends. The chat program confers special privileges on users in the buddy list, such as allowing them to send unsolicited messages to the user.

One way to implement the buddy lists for an online chat program is to store the information in a `multimap`. One `multimap` could store the buddy lists for every user. Each entry in the container stores one buddy for a user. The key is the user and the value is the buddy. For example, if Harry Potter and Ron Weasley had each other on their individual buddy lists, there would be two entries of the form “Harry Potter” maps to “Ron Weasley” and “Ron Weasley” maps to “Harry Potter.”

A multimap allows multiple values for the same key, so the same user is allowed multiple buddies. Here is the BuddyList class definition:

```
class BuddyList
{
public:
    // Adds buddy as a friend of name.
    void addBuddy(const std::string& name, const std::string& buddy);
    // Removes buddy as a friend of name
    void removeBuddy(const std::string& name, const std::string& buddy);
    // Returns true if buddy is a friend of name, false otherwise.
    bool isBuddy(const std::string& name, const std::string& buddy) const;
    // Retrieves a list of all the friends of name.
    std::list<std::string> getBuddies(const std::string& name) const;
private:
    std::multimap<std::string, std::string> mBuddies;
};
```

Here is the implementation, with comments explaining the code. It demonstrates the use of `lower_bound()`, `upper_bound()`, and `equal_range()`:

```
void BuddyList::addBuddy(const string& name, const string& buddy)
{
    // Make sure this buddy isn't already there. We don't want
    // to insert an identical copy of the key/value pair.
    if (!isBuddy(name, buddy)) {
        mBuddies.insert({ name, buddy }); // Using initializer_list
    }
}
void BuddyList::removeBuddy(const string& name, const string& buddy)
{
    // Obtain the beginning and end of the range of elements with
    // key 'name'. Use both lower_bound() and upper_bound() to demonstrate
    // their use. Otherwise, it's more efficient to call equal_range().
    auto iter = mBuddies.lower_bound(name); // Start of the range
    auto end = mBuddies.upper_bound(name); // End of the range
    // Iterate through the elements with key 'name' looking
    // for a value 'buddy'
    for (; iter != end; ++iter) {
        if (iter->second == buddy) {
            // We found a match! Remove it from the map.
            mBuddies.erase(iter);
            break;
        }
    }
}
bool BuddyList::isBuddy(const string& name, const string& buddy) const
{
    // Obtain the beginning and end of the range of elements with
    // key 'name' using equal_range().
    auto range = mBuddies.equal_range(name);
    auto iter = range.first; // Start of the range
    auto end = range.second; // End of the range
    // Iterate through the elements with key 'name' looking
    // for a value 'buddy'. If there are no elements with key 'name',
```

```
// iter equals end, so the loop body doesn't execute.
for (; iter != end; ++iter) {
    if (iter->second == buddy) {
        // We found a match!
        return true;
    }
}
// No matches
return false;
}
list<string> BuddyList::getBuddies(const string& name) const
{
    // Obtain the beginning and end of the range of elements with
    // key 'name' using equal_range().
    auto range = mBuddies.equal_range(name);
    auto iter = range.first; // Start of the range
    auto end = range.second; // End of the range
    // Create a list with all names in the range (all buddies of name).
    list<string> buddies;
    for (; iter != end; ++iter) {
        buddies.push_back(iter->second);
    }
    return buddies;
}
```

Note that `removeBuddy()` can't simply use the version of `erase()` that erases all elements with a given key, because it should erase only one element with the key, not all of them. Note also that `getBuddies()` can't use `insert()` on the list to insert the elements in the range returned by `equal_range()`, because the elements referred to by the `multimap` iterators are key/value pairs, not strings. The `getBuddies()` method must iterate explicitly through the list extracting the `string` from each key/value pair and pushing it onto the new list to be returned.

Here is a test of the `BuddyList`:

```
BuddyList buddies;
buddies.addBuddy("Harry Potter", "Ron Weasley");
buddies.addBuddy("Harry Potter", "Hermione Granger");
buddies.addBuddy("Harry Potter", "Hagrid");
buddies.addBuddy("Harry Potter", "Draco Malfoy");
// That's not right! Remove Draco.
buddies.removeBuddy("Harry Potter", "Draco Malfoy");
buddies.addBuddy("Hagrid", "Harry Potter");
buddies.addBuddy("Hagrid", "Ron Weasley");
buddies.addBuddy("Hagrid", "Hermione Granger");
auto harryBuds = buddies.getBuddies("Harry Potter");
cout << "Harry's friends: " << endl;
for (const auto& name : harryBuds) {
    cout << "\t" << name << endl;
}
```

The output is as follows:

```
Harry's friends:
    Ron Weasley
    Hermione Granger
    Hagrid
```

set

A `set`, defined in `<set>`, is very similar to a `map`. The difference is that instead of storing key/value pairs, in `sets` the value itself is the key. `sets` are useful for storing information in which there is no explicit key, but which you want to have in sorted order without any duplicates, with quick insertion, lookup, and deletion.

The interface supplied by `set` is almost identical to that of `map`. The main difference is that `set` doesn't provide `operator[]`.

You cannot change the key/value of elements in a `set` because modifying elements of a `set` while they are in the container would destroy the order.

set Example: Access Control List

One way to implement basic security on a computer system is through access control lists. Each entity on the system, such as a file or a device, has a list of users with permissions to access that entity. Users can generally be added to and removed from the permissions list for an entity only by users with special privileges. Internally, a `set` provides a nice way to represent the access control list. You could use one `set` for each entity, containing all the usernames who are allowed to access the entity. Here is a class definition for a simple access control list:

```
class AccessList
{
public:
    // Default constructor
    AccessList() = default;
    // Constructor to support uniform initialization.
    AccessList(const std::initializer_list<std::string>& initlst);
    // Adds the user to the permissions list.
    void addUser(const std::string& user);
    // Removes the user from the permissions list.
    void removeUser(const std::string& user);
    // Returns true if the user is in the permissions list.
    bool isAllowed(const std::string& user) const;
    // Returns a list of all the users who have permissions.
    std::list<std::string> getAllUsers() const;
private:
    std::set<std::string> mAllowed;
};
```

Here are the method definitions:

```
AccessList::AccessList(const initializer_list<string>& initlst)
{
    for (auto& user : initlst) {
        addUser(user);
    }
}
void AccessList::addUser(const string& user)
{
    mAllowed.insert(user);
}
```

```
void AccessList::removeUser(const string& user)
{
    mAllowed.erase(user);
}
bool AccessList::isAllowed(const string& user) const
{
    return (mAllowed.count(user) != 0);
}
list<string> AccessList::getAllUsers() const
{
    list<string> users;
    users.insert(end(users), begin(mAllowed), end(mAllowed));
    return users;
}
```

Finally, here is a simple test program:

```
AccessList fileX = { "pvw", "mgregoire", "baduser" };
fileX.removeUser("baduser");
if (fileX.isAllowed("mgregoire")) {
    cout << "mgregoire has permissions" << endl;
}
if (fileX.isAllowed("baduser")) {
    cout << "baduser has permissions" << endl;
}
auto users = fileX.getAllUsers();
for (const auto& user : users) {
    cout << user << " ";
}
```

The output of this program is as follows:

```
mgregoire has permissions
mgregoire  pvw
```

One of the constructors for `AccessList` uses an `initializer_list` as a parameter so that you can use the uniform initialization syntax, as demonstrated in the test program for initializing `fileX`.

multiset

A `multiset` is to a `set` what a `multimap` is to a `map`. A `multiset` supports all the operations of a `set`, but it allows multiple elements that are equal to each other to be stored in the container simultaneously. An example of a `multiset` is not shown because it's so similar to `set` and `multimap`.

UNORDERED ASSOCIATIVE CONTAINERS/ HASH TABLES

The STL has support for *unordered associative containers* or *hash tables*. There are four of them: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. The `map`, `multimap`, `set`, and `multiset` containers discussed earlier sort their elements, while these unordered variants do not sort their elements.

Hash Functions

The unordered associative containers are also called *hash tables*. That is because the implementation makes use of so called *hash functions*. The implementation usually consists of some kind of array where each element in the array is called a *bucket*. Each bucket has a specific numerical index like 0, 1, 2, up until the last bucket. A hash function transforms a key into a bucket index. The value associated with that key is then stored in that bucket. The result of a hash function is not always unique. The situation in which two or more keys hash to the same bucket index is called a *collision*. There are many approaches to handling collisions; for example, quadratic re-hashing, linear chaining, etc. Those who are interested may consult one of the references in the Algorithms and Data Structures section in Appendix B. The STL standard does not specify which collision-handling algorithm is required, but most current implementations have chosen to resolve collisions by linear chaining. With linear chaining, buckets do not directly contain the data values associated with the keys, but contain a pointer to a linked list. This linked list contains all the data values for that specific bucket. Figure 16-1 shows how this works.

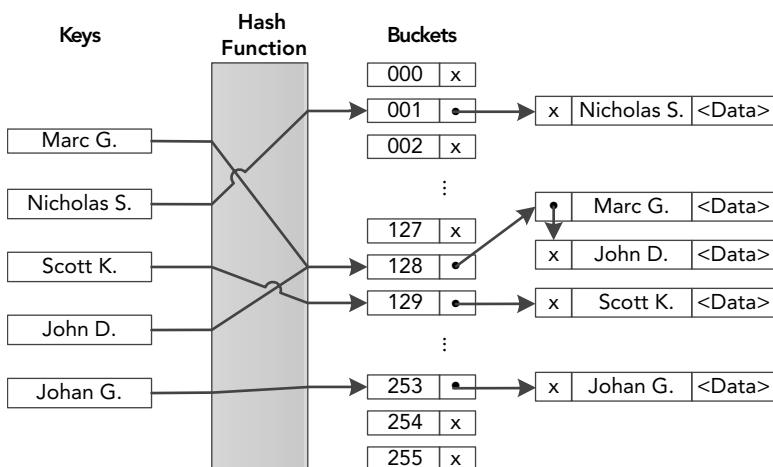


FIGURE 16-1

In Figure 16-1, applying the hash function to the keys “Marc G.” and “John D.” results in the same bucket index 128. This bucket then points to a linked list containing the keys “Marc G.” and “John D.” together with their associated data values. From Figure 16-1, it is also clear how lookups based on keys work and what the complexity is. A lookup involves a single hash function call to calculate the bucket index and after that one or more equality operations to find the right key in the linked list. This shows that lookups can be much faster compared to lookups with normal `maps`, but it all depends on how many collisions there are.

The choice of the hash function is very important. A hash function that creates no collisions is known as a “perfect hash.” A perfect hash has a lookup time that is constant; a regular hash has a lookup time that is, on average, close to 1, independent of the number of elements. As the number of

collisions increases, the lookup time increases, reducing performance. Collisions can be reduced by increasing the basic hash table size, but you need to take cache sizes into account.

The C++ standard provides hash functions for pointers and all primitive data types such as `bool`, `char`, `int`, `float`, `double`, and so on. Hash functions are also provided for `error_code`, `bitset`, `unique_ptr`, `shared_ptr`, `type_index`, `string`, `vector<bool>`, and `thread`. If there is no standard hash function available for the type of key you want to use, then you have to implement your own hash function. Creating a perfect hash is a nontrivial exercise, even when the set of keys is fixed and known. Creating one that is good enough for use with the unordered associative containers is not that hard.

The following example demonstrates how to write a custom hash function. This example simply forwards the request to one of the standard hash functions available. The code defines a class `IntWrapper` that just wraps a single integer. An `operator==` is provided because that's a requirement for keys used in unordered associative containers.

```
class IntWrapper
{
public:
    IntWrapper(int i) : mI(i) {}
    int getValue() const { return mI; }
    friend bool operator==(const IntWrapper& lhs, const IntWrapper& rhs);
private:
    int mI;
};
bool operator==(const IntWrapper& lhs, const IntWrapper& rhs)
{
    return lhs.mI == rhs.mI;
}
```

To write a hash function for `IntWrapper`, you write a specialization of the `std::hash` template for `IntWrapper`. The `std::hash` template is defined in `<functional>`. This specialization needs an implementation of the function call operator that calculates and returns the hash of a given `IntWrapper` instance. For this example, the request is simply forwarded to the standard hash function for integers:

```
namespace std
{
    template<> struct hash<IntWrapper>
    {
        typedef IntWrapper argument_type;
        typedef size_t result_type;
        result_type operator()(const argument_type& f) const {
            return std::hash<int>()(f.getValue());
        }
    };
}
```

Note that you normally are not allowed to put anything in the `std` namespace, however, `std` class template specializations are an exception to this rule. The two type definitions are required by the hash class template. The implementation of the function call operator is just one line. It creates an instance of the standard hash function for integers: `std::hash<int>()` and then calls the function call operator on it with an argument `f.getValue()`. Note that this forwarding works in this example because `IntWrapper` contains just one data member, an integer. If the class would contain multiple data members, then a hash needs to be calculated taking all these data members into account; however, these details fall outside the scope of this book.

unordered_map

unordered_map is defined in <unordered_map> as a class template:

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T>>>
class unordered_map;
```

There are five template parameters: the key type, the value type, the hash type, the equal comparison type, and the allocator type. With the last three parameters you can define your own hash function, equal comparison function, and allocator function, respectively. These parameters can usually be ignored because they have default values. I recommend you keep those default values when possible. The most important parameters are the first two. As with maps, uniform initialization can be used to initialize an unordered_map, as shown in the following example:

```
unordered_map<int, string> m = {
    {1, "Item 1"}, {2, "Item 2"}, {3, "Item 3"}, {4, "Item 4"}};
for (const auto& p : m) {
    cout << p.first << " = " << p.second << endl;
}
```

The following table summarizes the differences between a map and an unordered_map.

OPERATION	map	unordered_map
at()	x	x
begin()	x	x
bucket()		x
bucket_count()		x
bucket_size()		x
cbegin()	x	x
cend()	x	x
clear()	x	x
count()	x	x
crbegin()	x	
crend()	x	

continues

(continued)

OPERATION	map	unordered_map
emplace()	x	x
emplace_hint()	x	x
empty()	x	x
end()	x	x
equal_range()	x	x
erase()	x	x
find()	x	x
insert()	x	x
iterator / const_iterator	x	x
load_factor()		x
local_iterator / const_local_iterator		x
lower_bound()	x	
max_bucket_count()		x
max_load_factor()		x
max_size()	x	x
operator[]	x	x
rbegin()	x	
rehash()		x
rend()	x	
reserve()		x
reverse_iterator / const_reverse_iterator	x	
size()	x	x
swap()	x	x
upper_bound()	x	

As with a normal `map`, all keys in an `unordered_map` should be unique. The preceding table includes a number of hash-specific methods. For example, `load_factor()` returns the average number of elements per bucket to give you an indication on the number of collisions. The `bucket_count()` method returns the number of buckets in the container. It also provides a `local_iterator` and `const_local_iterator` allowing you to iterate over the elements in a single bucket; but, these may

not be used to iterate across buckets. The `bucket(key)` method returns the index of the bucket that contains the given key; `begin(n)` returns a `local_iterator` referring to the first element in the bucket with index `n`, and `end(n)` returns a `local_iterator` referring to one-past-the-last element in the bucket with index `n`. The example in the next section demonstrates how to use these methods.

unordered_map Example: Phone Book

The following example uses an `unordered_map` to represent a phone book. The name of a person is the key while the phone number is the value associated with that key.

```

template<class T>
void printMap(const T& m)
{
    for (auto& p : m) {
        cout << p.first << " (Phone: " << p.second << ")" << endl;
    }
    cout << "-----" << endl;
}
int main()
{
    // Create a hash table.
    unordered_map<string, string> um = {
        { "Marc G.", "123-456789" },
        { "Scott K.", "654-987321" } };
    printMap(um);

    // Add/remove some phone numbers.
    um.insert(make_pair("John D.", "321-987654"));
    um["Johan G."] = "963-258147";
    um["Freddy K."] = "999-256256";
    um.erase("Freddy K.");
    printMap(um);

    // Find the bucket index for a specific key.
    int bucket = um.bucket("Marc G.");
    cout << "Marc G. is in bucket " << bucket
        << " which contains the following "
        << um.bucket_size(bucket) << " elements:" << endl;
    // Get begin and end iterators for the elements in this bucket.
    // 'auto' is being used here. The compiler will derive the type
    // of both iterators as unordered_map<string, string>::const_local_iterator
    auto liter = um.cbegin(bucket);
    auto literEnd = um.cend(bucket);
    while (liter != literEnd) {
        cout << "\t" << liter->first << " (Phone: " << liter->second << ")" << endl;
        ++liter;
    }
    cout << "-----" << endl;

    // Print some statistics about the hash table
    cout << "There are " << um.bucket_count() << " buckets." << endl;
    cout << "Average number of elements in a bucket is " << um.load_factor() << endl;

    return 0;
}

```

A possible output is as follows. Note that the output can be different on a different system because it depends on the implementation of the hash functions in the STL being used.

```
Scott K. (Phone: 654-987321)
Marc G. (Phone: 123-456789)
-----
Scott K. (Phone: 654-987321)
Marc G. (Phone: 123-456789)
Johan G. (Phone: 963-258147)
John D. (Phone: 321-987654)
-----
Marc G. is in bucket 1 which contains the following 2 elements:
  Scott K. (Phone: 654-987321)
  Marc G. (Phone: 123-456789)
-----
There are 8 buckets.
Average number of elements in a bucket is 0.5
```

unordered_multimap

An `unordered_multimap` is an `unordered_map` that allows multiple elements with the same key. Their interfaces are almost identical, with the following changes:

- `unordered_multimaps` do not provide `operator[]`. The semantics of this operator do not make sense if there can be multiple elements with a single key.
- Inserts on `unordered_multimaps` always succeed. Thus, the `unordered_multimap::insert()` method that adds a single element returns only an iterator.

NOTE `unordered_multimaps` allow you to insert identical key/value pairs. If you want to avoid this redundancy, you must check explicitly before inserting a new element.

As discussed earlier with `multimaps`, looking up elements in `unordered_multimaps` cannot be done using `operator[]` because it is not provided. You can use `find()` but it returns an iterator referring to any one of the elements with a given key (not necessarily the first element with that key). Instead, it's best to use the `equal_range()` method, which returns a pair of iterators: one referring to the first element matching a given key and one referring to one-past-the-last element matching a given key. The use of `equal_range()` is exactly the same as discussed for `multimaps`, so you can look at the example given for `multimaps` to see how it works.

unordered_set/unordered_multiset

The `<unordered_set>` header file defines `unordered_set` and `unordered_multiset`, which are very similar to `set` and `multiset`, respectively; except that they do not sort their keys and that they use a hash function. The differences between `unordered_set` and `unordered_map` are similar to the differences between `set` and `map` as discussed earlier in this chapter, so they are not discussed in detail here. Consult a Standard Library Reference for a thorough summary of `unordered_set` and `unordered_multiset` operations.

OTHER CONTAINERS

There are several other parts of the C++ language that work with the STL to varying degrees, including standard C-style arrays, strings, streams, and `bitset`.

Standard C-Style Arrays

Recall that “dumb” pointers are bona fide iterators because they support the required operators. This point is more than just a piece of trivia. It means that you can treat standard C-style arrays as STL containers by using pointers to their elements as iterators. Standard C-style arrays, of course, don’t provide methods like `size()`, `empty()`, `insert()`, and `erase()`, so they aren’t true STL containers. Nevertheless, because they do support iterators through pointers, you can use them in the algorithms described in Chapter 17 and in some of the methods described in this chapter.

For example, you could copy all the elements of a standard C-style array into a `vector` using the `insert()` method of a `vector` that takes an iterator range from any container. This `insert()` method prototype looks like this:

```
template <class InputIterator> iterator insert(const_iterator position,
                                              InputIterator first, InputIterator last);
```

If you want to use a standard C-style `int` array as the source, then the templated type of `InputIterator` becomes `int*`. Here is a full example:

```
const size_t count = 10;
unsigned int arr[count];      // standard C-style array
// Initialize each element of the array to the value of its index.
for (unsigned int i = 0; i < count; i++) {
    arr[i] = i;
}
vector<int> vec;      // STL vector
// Insert the contents of the array at the end of the vector.
vec.insert(end(vec), arr, arr + count);
// Print the contents of the vector.
for (const auto& i : vec) {
    cout << i << " ";
}
```

Note that the iterator referring to the first element of the array is the address of the first element, which is `arr` in this case. The name of an array alone is interpreted as the address of the first element. The iterator referring to the end must be one-past-the-last element, so it’s the address of the first element plus 10, or `arr+10`.

It’s easier to use `std::begin()` or `std::cbegin()` to get an iterator to the first element of a stack-based array, and `std::end()` or `std::cend()` to get an iterator to one-past-the-last element of a stack-based array. For example, the call to `insert()` in the previous example can be written as follows:

```
vec.insert(end(vec), cbegin(arr), cend(arr));
```

WARNING Functions such as `std::begin()` and `std::end()` only work on stack-based C-style arrays, not on heap-based C-style arrays.

strings

You can think of a `string` as a sequential container of characters. Thus, it shouldn't be surprising to learn that a C++ `string` is a full-fledged sequential container. It contains `begin()` and `end()` methods that return iterators into the `string`, `insert()`, `push_back()` and `erase()` methods, `size()`, `empty()`, and all the rest of the sequential container basics. It resembles a `vector` quite closely, even providing methods `reserve()` and `capacity()`.

You can use `string` as an STL container just as you would use `vector`. Here is an example:

```
string str1;
str1.insert(cend(str1), 'h');
str1.insert(cend(str1), 'e');
str1.push_back('l');
str1.push_back('l');
str1.push_back('o');
for (const auto& letter : str1) {
    cout << letter;
}
cout << endl;
for (auto it = cbegin(str1); it != cend(str1); ++it) {
    cout << *it;
}
cout << endl;
```

In addition to the STL sequential container methods, `strings` provide a whole host of useful methods and `friend` functions. The `string` interface is actually quite a good example of a cluttered interface, one of the design pitfalls discussed in Chapter 6. The `string` class is discussed in detail in Chapter 2.

Streams

Input and output streams are not containers in the traditional sense: they do not store elements. However, they can be considered sequences of elements, and as such share some characteristics with STL containers. C++ streams do not provide any STL-related methods directly, but the STL supplies special iterators called `istream_iterator` and `ostream_iterator` that allow you to “iterate” through input and output streams. Chapter 20 explains how to use them.

bitset

A `bitset` is a fixed-length abstraction of a sequence of bits. A bit can represent only two values, 1 and 0, which can be referred to as on/off, true/false, etc. A `bitset` also uses the terminology `set` and `unset`. You can `toggle` or `flip` a bit from one value to the other.

A `bitset` is not a true STL container: it's of fixed size, it's not templated on an element type, and it doesn't support iteration. However, it's a useful utility, which is often lumped with the containers, so a brief introduction is provided here. Consult a Standard Library Reference for a thorough summary of the `bitset` operations.

bitset Basics

A `bitset`, defined in `<bitset>`, is templated on the number of bits it stores. The default constructor initializes all fields of a `bitset` to 0. An alternative constructor creates a `bitset` from a `string` of 0 and 1 characters.

You can adjust the values of individual bits with the `set()`, `reset()`, and `flip()` methods, and you can access and set individual fields with an overloaded `operator[]`. Note that `operator[]` on a non-const object returns a proxy object to which you can assign a Boolean value, call `flip()`, or complement with `~`. You can also access individual fields with the `test()` method. Additionally, you can stream `bitsets` with the normal insertion and extraction operators. A `bitset` is streamed as a `string` of 0 and 1 characters.

Here is a small example:

```
bitset<10> myBitset;
myBitset.set(3);
myBitset.set(6);
myBitset[8] = true;
myBitset[9] = myBitset[3];
if (myBitset.test(3)) {
    cout << "Bit 3 is set!" << endl;
}
cout << myBitset << endl;
```

The output is as follows:

```
Bit 3 is set!
1101001000
```

Note that the leftmost character in the output `string` is the highest numbered bit. This corresponds to our intuitions about binary number representations, where the low-order bit representing $2^0 = 1$ is the rightmost bit in the printed representation.

Bitwise Operators

In addition to the basic bit manipulation routines, a `bitset` provides implementations of all the bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>`, `&=`, `|=`, `^=`, `<<=`, and `>>=`. They behave just as they would on a “real” sequence of bits. Here is an example:

```
auto str1 = "0011001100";
auto str2 = "0000111100";
bitset<10> bitsOne(str1);
bitset<10> bitsTwo(str2);
auto bitsThree = bitsOne & bitsTwo;
cout << bitsThree << endl;
```

```
bitsThree <= 4;
cout << bitsThree << endl;
```

The output of the program is as follows:

```
0000001100
0011000000
```

bitset Example: Representing Cable Channels

One possible use of bitsets is tracking channels of cable subscribers. Each subscriber could have a bitset of channels associated with his or her subscription, with set bits representing the channels to which he or she actually subscribes. This system could also support “packages” of channels, also represented as bitsets, which represent commonly subscribed combinations of channels.

The following `CableCompany` class is a simple example of this model. It uses two `maps`, each of `string/bitset`, storing the cable packages as well as the subscriber information.

```
const size_t kNumChannels = 10;
class CableCompany
{
public:
    // Adds the package with the specified channels to the database.
    void addPackage(const std::string& packageName,
                    const std::bitset<kNumChannels>& channels);
    // Removes the specified package from the database
    void removePackage(const std::string& packageName);
    // Adds customer to database with initial channels found in package.
    // Throws out_of_range if the package name is invalid.
    // Throws invalid_argument if the customer is already known.
    void newCustomer(const std::string& name, const std::string& package);
    // Adds customer to database with given initial channels.
    // Throws invalid_argument if the customer is already known.
    void newCustomer(const std::string& name,
                     const std::bitset<kNumChannels>& channels);
    // Adds the channel to the customers profile.
    // Throws invalid_argument if the customer is unknown.
    void addChannel(const std::string& name, int channel);
    // Removes the channel from the customers profile.
    // Throws invalid_argument if the customer is unknown.
    void removeChannel(const std::string& name, int channel);
    // Adds the specified package to the customers profile.
    // Throws out_of_range if the package name is invalid.
    // Throws invalid_argument if the customer is unknown.
    void addPackageToCustomer(const std::string& name,
                             const std::string& package);
    // Removes the specified customer from the database.
    void deleteCustomer(const std::string& name);
    // Retrieves the channels to which a customer subscribes.
    // Throws invalid_argument if the customer is unknown.
    std::bitset<kNumChannels>& getCustomerChannels(const std::string& name);
private:
    typedef std::map<std::string, std::bitset<kNumChannels>> MapType;
```

```
    MapType mPackages, mCustomers;
};
```

Here are the implementations of all methods, with comments explaining the code:

```
void CableCompany::addPackage(const string& packageName,
    const bitset<kNumChannels>& channels)
{
    // Just make a key/value pair and insert it into the packages map.
    mPackages.insert({ packageName, channels });
}
void CableCompany::removePackage(const string& packageName)
{
    // Just erase the package from the package map
    mPackages.erase(packageName);
}
void CableCompany::newCustomer(const string& name, const string& package)
{
    // Get a reference to the specified package.
    auto it = mPackages.find(package);
    if (it == end(mPackages)) {
        // That package doesn't exist. Throw an exception.
        throw out_of_range("Invalid package");
    }
    // Create the account with the bitset representing that package.
    // Note that 'it' refers to a name/bitset pair. The bitset is the
    // second field.
    auto result = mCustomers.insert({ name, it->second });
    if (!result.second) {
        // Customer was already in the database. Nothing changed.
        throw invalid_argument("Duplicate customer");
    }
}
void CableCompany::newCustomer(const string& name,
    const bitset<kNumChannels>& channels)
{
    // Just add the customer/channels pair to the customers map.
    auto result = mCustomers.insert({ name, channels });
    if (!result.second) {
        // Customer was already in the database. Nothing changed.
        throw invalid_argument("Duplicate customer");
    }
}
void CableCompany::addChannel(const string& name, int channel)
{
    // Find a reference to the customer.
    auto it = mCustomers.find(name);
    if (it != end(mCustomers)) {
        // We found this customer; set the channel.
        // Note that 'it' is a reference to a name/bitset pair.
        // The bitset is the second field.
        it->second.set(channel);
    } else {
        throw invalid_argument("Unknown customer");
    }
}
```

```
}

void CableCompany::removeChannel(const string& name, int channel)
{
    // Find a reference to the customer.
    auto it = mCustomers.find(name);
    if (it != end(mCustomers)) {
        // We found this customer; remove the channel.
        // Note that 'it' is a reference to a name/bitset pair.
        // The bitset is the second field.
        it->second.reset(channel);
    } else {
        throw invalid_argument("Unknown customer");
    }
}

void CableCompany::addPackageToCustomer(const string& name,
                                         const string& package)
{
    // Find the package.
    auto itPack = mPackages.find(package);
    if (itPack == end(mPackages)) {
        // That package doesn't exist. Throw an exception.
        throw out_of_range("Invalid package");
    }

    // Find the customer.
    auto itCust = mCustomers.find(name);
    if (itCust != end(mCustomers)) {
        // Or-in the package to the customers existing channels.
        // Note that the iterators are references to name/bitset pairs.
        // The bitset is the second field.
        itCust->second |= itPack->second;
    } else {
        throw invalid_argument("Unknown customer");
    }
}

void CableCompany::deleteCustomer(const string& name)
{
    // Remove the customer with this name
    mCustomers.erase(name);
}

bitset<kNumChannels>& CableCompany::getCustomerChannels(const string& name)
{
    // Find the customer
    auto it = mCustomers.find(name);
    if (it != end(mCustomers)) {
        // Found it.
        // Note that 'it' is a reference to a name/bitset pair.
        // The bitset is the second field.
        return it->second;
    }
    // Didn't find it. Throw an exception.
    throw invalid_argument("Unknown customer");
}
```

Finally, here is a simple program demonstrating how to use the `CableCompany` class:

```
CableCompany myCC;
auto basic_pkg = "1111000000";
auto premium_pkg = "1111111111";
auto sports_pkg = "0000100111";
myCC.addPackage("basic", bitset<kNumChannels>(basic_pkg));
myCC.addPackage("premium", bitset<kNumChannels>(premium_pkg));
myCC.addPackage("sports", bitset<kNumChannels>(sports_pkg));
myCC.newCustomer("Marc G.", "basic");
myCC.addPackageToCustomer("Marc G.", "sports");
cout << myCC.getCustomerChannels("Marc G.") << endl;
```

SUMMARY

This chapter introduced the standard template library containers. It presented sample code illustrating a variety of uses for these containers. Hopefully, you appreciate the power of `vector`, `deque`, `list`, `forward_list`, `array`, `stack`, `queue`, `priority_queue`, `map`, `multimap`, `set`, `multiset`, `unordered_map`, `unordered_multimap`, `unordered_set`, `unordered_multiset`, `string`, and `bitset`. Even if you don't incorporate them into your programs immediately, at least keep them in the back of your mind for future projects.

Now that you are familiar with the containers, the next chapter can illustrate the true power of the STL with a discussion on generic algorithms.

17

Mastering STL Algorithms

WHAT'S IN THIS CHAPTER?

- Algorithms explained
- Lambda expressions explained
- Function objects explained
- The details of the STL algorithms
- A larger example: auditing voter registrations

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

As Chapter 16 shows, the STL provides an impressive collection of generic data structures. Most libraries stop there. The STL, however, contains an additional assortment of generic algorithms that can, with some exceptions, be applied to elements from any container. Using these algorithms, you can find, sort, and process elements in containers, and perform a whole host of other operations. The beauty of the algorithms is that they are independent not only of the types of the underlying elements, but also of the types of the containers on which they operate. Algorithms perform their work using only the iterator interfaces.

Many of the algorithms accept a *callback*, which can be a function pointer or something that behaves like a function pointer, such as an object with an overloaded `operator()` or an inline lambda expression. Conveniently, the STL provides a set of classes that can be used to create callback objects for the algorithms. These callback objects are called function objects, or just *functors*.

OVERVIEW OF ALGORITHMS

The “magic” behind the algorithms is that they work on iterator intermediaries instead of on the containers themselves. In that way, they are not tied to specific container implementations. All the STL algorithms are implemented as function templates, where the template type parameters are usually iterator types. The iterators themselves are specified as arguments to the function. Templatized functions can usually deduce the template types from the function arguments, so you can generally call the algorithms as if they were normal functions, not templates.

The iterator arguments are usually iterator ranges. As Chapter 16 explains, iterator ranges are half-open for most containers such that they include the first element in the range, but exclude the last. The end iterator is really a “past-the-end” marker.

Algorithms pose certain requirements on iterators passed to it. For instance, `copy_backward()` is an example of an algorithm that requires a bidirectional iterator, and `stable_sort()` is an example of an algorithm requiring random access iterators. This means that such algorithms cannot work on containers that do not provide the necessary iterators. `forward_list` is an example of a container supporting only forward iterators, no bidirectional or random access iterators, thus `copy_backward()` and `stable_sort()` cannot work on `forward_list`.

Most algorithms are defined in the `<algorithm>` header file, while some numerical algorithms are defined in `<numeric>`. All of them are in the `std` namespace.

The best way to understand the algorithms is to look at some examples first. After you’ve seen how a few of them work, it’s easy to pick up the others. This section describes the `find()`, `find_if()`, and `accumulate()` algorithms in detail. The next sections present the lambda expressions and function objects, and discusses each of the classes of algorithms with representative samples.

The `find` and `find_if` Algorithms

`find()` looks for a specific element in an iterator range. You can use it on elements in any container type. It returns an iterator referring to the element found, or the end iterator of the range in case the element is not found. Note that the range specified in the call to `find()` need not be the entire range of elements in a container; it could be a subset.

WARNING *If `find()` fails to find an element, it returns an iterator equal to the end iterator specified in the function call, not the end iterator of the underlying container.*

Here is an example of `find()`. Note that this example assumes that the user plays nice and enters valid numbers; it does not perform any error checking on the user input. Performing error checking on stream input is discussed in Chapter 12.

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
int main()
{
```

```

int num;
vector<int> myVector;
while (true) {
    cout << "Enter a number to add (0 to stop): ";
    cin >> num;
    if (num == 0) {
        break;
    }
    myVector.push_back(num);
}
while (true) {
    cout << "Enter a number to lookup (0 to stop): ";
    cin >> num;
    if (num == 0) {
        break;
    }
    auto endIt = cend(myVector);
    auto it = find(cbegin(myVector), endIt, num);
    if (it == endIt) {
        cout << "Could not find " << num << endl;
    } else {
        cout << "Found " << *it << endl;
    }
}
return 0;
}

```

The call to `find()` is made with `cbegin(myVector)` and `endIt` as arguments, where `endIt` is defined as `cend(myVector)` in order to search all the elements of the `vector`. If you want to search in a sub range, you can change these two iterators.

Here is a sample run of the program:

```

Enter a number to add (0 to stop): 3
Enter a number to add (0 to stop): 4
Enter a number to add (0 to stop): 5
Enter a number to add (0 to stop): 6
Enter a number to add (0 to stop): 0
Enter a number to lookup (0 to stop): 5
Found 5
Enter a number to lookup (0 to stop): 8
Could not find 8
Enter a number to lookup (0 to stop): 0

```

Some containers, such as `map` and `set`, provide their own versions of `find()` as class methods.

WARNING If a container provides a method with the same functionality as a generic algorithm, you should use the method instead, because it's faster. For example, the generic `find()` algorithm runs in linear time, even on a `map`, while the `find()` method on a `map` runs in logarithmic time.

`find_if()` is similar to `find()`, except that it accepts a *predicate function callback* instead of a simple element to match. A predicate returns `true` or `false`. The `find_if()` algorithm calls the

predicate on each element in the range until the predicate returns `true`, in which case `find_if()` returns an iterator referring to that element. The following program reads test scores from the user, then checks if any of the scores are “perfect.” A perfect score is a score of 100 or higher. The program is similar to the previous example. Only the differences are highlighted.

```
bool perfectScore(int num)
{
    return (num >= 100);
}
int main()
{
    int num;
    vector<int> myVector;
    while (true) {
        cout << "Enter a test score to add (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        myVector.push_back(num);
    }
    auto endIt = cend(myVector);
    auto it = find_if(cbegin(myVector), endIt, perfectScore);
    if (it == endIt) {
        cout << "No perfect scores" << endl;
    } else {
        cout << "Found a \"perfect\" score of " << *it << endl;
    }
    return 0;
}
```

This program passes a pointer to the `perfectScore()` function, which the `find_if()` algorithm then calls on each element until it returns `true`.

Here is the same example but using a lambda expression. It gives you an initial idea about the power of lambda expressions. Don’t worry about their syntax; they are explained in detail later in this chapter. Note the absence of the `perfectScore()` function.

```
int num;
vector<int> myVector;
while (true) {
    cout << "Enter a test score to add (0 to stop): ";
    cin >> num;
    if (num == 0) {
        break;
    }
    myVector.push_back(num);
}
auto endIt = cend(myVector);
auto it = find_if(cbegin(myVector), endIt, [] (int i){ return i >= 100; });
if (it == endIt) {
    cout << "No perfect scores" << endl;
} else {
    cout << "Found a \"perfect\" score of " << *it << endl;
}
```

Unfortunately, the STL provides no `find_all()` or equivalent algorithm that returns all instances matching a predicate. Chapter 20 shows you how to write your own `find_all()` algorithm.

The accumulate Algorithms

It's often useful to calculate the sum, or some other arithmetic quantity, of all the elements in a container. The `accumulate()` function does just that. In its most basic form, it calculates the sum of the elements in a specified range. For example, the following function calculates the arithmetic mean of a sequence of integers in a `vector`. The arithmetic mean is simply the sum of all the elements divided by the number of elements.

```
#include <numeric>
#include <vector>
using namespace std;
double arithmeticMean(const vector<int>& nums)
{
    double sum = accumulate(cbegin(nums), cend(nums), 0);
    return sum / nums.size();
}
```

Note that `accumulate()` is declared in `<numeric>`, not in `<algorithm>`. The `accumulate()` algorithm takes as its third parameter an initial value for the sum, which in this case should be 0 (the identity for addition) to start a fresh sum.

The second form of `accumulate()` allows the caller to specify an operation to perform instead of the default addition. This operation takes the form of a binary callback. Suppose that you want to calculate the geometric mean, which is the product of all the numbers in the sequence to the power of the inverse of the size. In that case, you would want to use `accumulate()` to calculate the product instead of the sum. You *could* write it like this:

```
#include <numeric>
#include <vector>
#include <cmath>
using namespace std;
int product(int num1, int num2)
{
    return num1 * num2;
}
double geometricMean(const vector<int>& nums)
{
    double mult = accumulate(cbegin(nums), cend(nums), 1, product);
    return pow(mult, 1.0 / nums.size());
}
```

Note that the `product()` function is passed as a callback to `accumulate()` and that the initial value for the accumulation is 1 (the identity for multiplication) instead of 0.

To give you a second teaser about the power of lambda expressions, the `geometricMean()` function *could* be written as follows, without using the `product()` function:

```
double geometricMeanLambda(const vector<int>& nums)
{
    double mult = accumulate(cbegin(nums), cend(nums), 1,
```

```
 [] (int num1, int num2){ return num1 * num2; });
return pow(mult, 1.0 / nums.size());
}
```

Later in this chapter you learn how to use `accumulate()` in the `geometricMean()` function without writing a function callback or lambda expression.

Move Semantics with Algorithms

Just like STL containers, STL algorithms are also optimized to use move semantics at appropriate times. This can greatly speed up certain algorithms; for example, `sort()`. For this reason, it is highly recommended that you implement move semantics in your custom element classes that you want to store in containers. Move semantics can be added to any class by implementing a move constructor and a move assignment operator. Both should be marked as `noexcept`, because they should not throw exceptions. Consult the “Move Semantics” section in Chapter 10 for details on how to add move semantics to your classes.

LAMBDA EXPRESSIONS

Lambda expressions allow you to write anonymous functions inline, removing the need to write a separate function or a function object. Lambda expressions can make code easier to read and understand.

Syntax

Let’s start with a very simple lambda expression. The following example defines a lambda expression that just writes a string to the console. A lambda expression starts with square brackets `[]`, followed by curly braces `{}`, which contain the body of the lambda expression. The lambda expression is assigned to the `basicLambda` auto-typed variable. The second line executes the lambda expression using normal function call syntax.

```
auto basicLambda = []{ cout << "Hello from Lambda" << endl; };
basicLambda();
```

The output is as follows:

```
Hello from Lambda
```

A lambda expression can accept a parameter. Parameters are specified between parentheses and separated by commas, just as with normal functions. Here is an example using one parameter:

```
auto parametersLambda =
[] (int value){ cout << "The value is " << value << endl; };
parametersLambda(42);
```

If a lambda expression does not accept any parameters, you can either specify empty parentheses or simply omit them.

A lambda expression can return a value. The return type is specified following an arrow, called a trailing return type. The following example defines a lambda expression accepting two parameters and returning the sum:

```
auto returningLambda = [](int a, int b) -> int { return a + b; };
int sum = returningLambda(11, 22);
```

The return type can be omitted even if the lambda expression does return something. If the return type is omitted, the compiler deduces the return type of the lambda expression according to the same rules as for function return type deduction (see Chapter 1). In the previous example, the return type can be omitted as follows:

```
auto returningLambda = [](int a, int b){ return a + b; };
int sum = returningLambda(11, 22);
```

A lambda expression can capture variables from its enclosing scope. For example, the following lambda expression captures the variable `data` so that it can be used in its body.

```
double data = 1.23;
auto capturingLambda = [data]{ cout << "Data = " << data << endl; };
```

The square brackets part is called the lambda *capture block*. It allows you to specify how you want to *capture* variables from the enclosing scope. *Capturing* a variable means that the variable becomes available inside the body of the lambda expression. Specifying an empty capture block, `[]`, means that no variables from the enclosing scope are captured. When you just write the name of a variable in the capture block as in the preceding example, then you are capturing that variable by value.

The compiler transforms a lambda expression into some kind of functor. Chapter 14 explains what a functor is. The captured variables become data members of this functor. Variables captured by value are copied into data members of the functor. These data members have the same constness as the captured variables. In the preceding `capturingLambda` example, the functor gets a non-const data member called `data`, because the captured variable, `data`, is non-const. However, in the following example, the functor gets a const data member called `data`, because the captured variable is const.

```
const double data = 1.23;
auto capturingLambda = [data]{ cout << "Data = " << data << endl; };
```

A functor always has an implementation of the function call operator. For a lambda expression, this function call operator is marked as `const` by default. That means that even if you capture a non-const variable by value in a lambda expression, the lambda expression will not be able to modify the copy. You can mark the function call operator as non-const by specifying the lambda expression as `mutable` as follows:

```
double data = 1.23;
auto capturingLambda =
    [data] () mutable { data *= 2; cout << "Data = " << data << endl; };
```

In this example, the non-const variable `data` is captured by value, thus the functor gets a non-const data member that is a copy of `data`. Because of the `mutable` keyword, the function call operator is marked as non-const, thus the body of the lambda expression can modify its copy of `data`. Note that if you specify `mutable`, then you have to specify the parentheses for the parameters even if they are empty.

You can prefix the name of a variable with `&` to capture it by reference. The following example captures the variable `data` by reference so that the lambda expression can directly change `data` in the enclosing scope:

```
double data = 1.23;
auto capturingLambda = [&data]{ data *= 2; };
```

When you capture a variable by reference, you have to make sure that the reference is still valid at the time the lambda expression is executed.

There are two ways to capture all variables from the enclosing scope:

- [=] captures all variables by value
- [&] captures all variables by reference

It is also possible to selectively decide which variables to capture and how, by specifying a *capture list* with an optional *capture default*. Variables prefixed with & are captured by reference. Variables without a prefix are captured by value. The capture default should be the first element in the capture list and be either & or =. Here are some capture block examples:

- [&x] captures only x by reference and nothing else.
- [x] captures only x by value and nothing else.
- [=, &x, &y] captures by value by default, except variables x and y, which are captured by reference.
- [&, x] captures by reference by default, except variable x, which is captured by value.
- [&x, &x] is illegal because identifiers cannot be repeated.
- [this] captures the surrounding object. In the body of the lambda expression you can access this object, even without using this->.

WARNING *It is not recommended to capture all variables from the enclosing scope with [=], [&], or with a capture default, even though it is possible. Instead, you should selectively capture only what's needed.*

Full Syntax

The complete syntax of a lambda expression is as follows:

```
[capture_block] (parameters) mutable exception_specification attribute_specifier
    -> return_type {body}
```

A lambda expression contains the following parts:

- **Capture block:** specifies how variables from the enclosing scope are captured and made available in the body of the lambda.
- **Parameters:** (optional) a list of parameters for the lambda expression. You can omit this list only if you do not need any parameters and you do not specify `mutable`, an exception specification, attribute specifier, or a return type. The parameter list is similar to the parameter list for normal functions.
- **Mutable:** (optional) marks the lambda expression as mutable; see previous section.
- **exception_specification:** (optional) can be used to specify which exceptions the body of the lambda expression can throw.

- **attribute_specifier:** (optional) can be used to specify attributes for the lambda expression. Attributes are explained in Chapter 10.
- **return_type:** (optional) the type of the returned value. If this is omitted, the compiler deduces the return type according to the same rules as for function return type deduction; see Chapter 1.



Generic Lambda Expressions

Starting with C++14 it is possible to use auto type deduction for parameters of lambda expressions instead of explicitly specifying concrete types for them. To specify auto type deduction for a parameter, the type is simply specified as `auto`. The type deduction rules are the same as those for template argument deduction.

The following example defines a generic lambda expression called `isGreaterThan100`. This single lambda expression is used with the `find_if()` algorithm, once for a vector of integers and once for a vector of doubles.

```
vector<int> ints{ 11, 55, 101, 200 };
vector<double> doubles{ 11.1, 55.5, 200.2 };

// Define a generic lambda to find values > 100.
auto isGreaterThan100 = [](auto i){ return i > 100; };

// Use the generic lambda with the vector of integers.
auto it1 = find_if(cbegin(ints), cend(ints), isGreaterThan100);
if (it1 != cend(ints)) {
    cout << "Found a value > 100: " << *it1 << endl;
}

// Use exactly the same generic lambda with the vector of doubles.
auto it2 = find_if(cbegin(doubles), cend(doubles), isGreaterThan100);
if (it2 != cend(doubles)) {
    cout << "Found a value > 100: " << *it2 << endl;
}
```



Lambda Capture Expressions

C++14 adds support for *lambda capture expressions* to initialize capture variables with any kind of expression. It can be used to introduce variables in the lambda expression that are not at all captured from the enclosing scope. For example, the following code creates a lambda expression. Inside this lambda expression there are two variables available; one called `myCapture`, initialized to the string “Pi:” using a lambda capture expression, and one called `pi`, which is captured by value from the enclosing scope. Note that non-reference capture variables such as `myCapture` that are initialized with a capture initializer are copy constructed, which means that `const` qualifiers are stripped.

```
double pi = 3.1415;
auto myLambda = [myCapture = "Pi: ", pi]{ std::cout << myCapture << pi; };
```

A lambda capture variable can be initialized with any kind of expression, and also with `std::move()`. This is important for objects that cannot be copied, only moved; for example, `unique_ptr`. By default, capturing by value uses copy semantics, so it's impossible to capture a

`unique_ptr` by value in a lambda expression. Using a lambda capture expression, it is possible to capture it by moving. For example:

```
auto myPtr = std::make_unique<double>(3.1415);
auto myLambda = [p = std::move(myPtr)]{ std::cout << *p; };
```

It's allowed to have the same name for the capture variable as the name in the enclosing scope. The previous example can be written as follows:

```
auto myPtr = std::make_unique<double>(3.1415);
auto myLambda = [myPtr = std::move(myPtr)]{ std::cout << *myPtr; };
```

Lambda Expressions as Return Type

`std::function` defined in the `<functional>` header file is a *polymorphic function wrapper* and is similar to a function pointer. It can be bound to anything that can be called (functors, member function pointers, function pointers, and lambdas) as long as the arguments and return type are compatible with those of the wrapper. A wrapper for a function that returns a `double` and takes two integers as parameters can be defined as follows:

```
function<double(int, int)> myWrapper;
```

By using `std::function`, lambda expressions can be given a name and can be returned from functions. Take a look at the following definition:

```
function<int(void)> multiplyBy2Lambda(int x)
{
    return [x]{ return 2*x; };
```

The body of this function creates a lambda expression that captures the variable `x` from the enclosing scope by value and returns an integer that is two times the value passed to `multiplyBy2Lambda()`. The return type of the `multiplyBy2Lambda()` function is `function<int(void)>`, which is a function accepting no arguments and returning an integer. The lambda expression defined in the body of the function exactly matches this prototype. The variable `x` is captured by value and thus a copy of the value of `x` is bound to the `x` in the lambda expression before the lambda is returned from the function.

C++14 supports function return type deduction (see Chapter 1), which allows you to write the `multiplyBy2Lambda()` function as follows:

```
auto multiplyBy2Lambda(int x)
{
    return [x]{ return 2 * x; };
```

The function can be called as follows:

```
function<int(void)> fn = multiplyBy2Lambda(5);
cout << fn() << endl;
```

You can use the `auto` keyword to make this much easier:

```
auto fn = multiplyBy2Lambda(5);
cout << fn() << endl;
```

The output will be 10.

The `multiplyBy2Lambda()` example captures the variable `x` by value, `[x]`. Suppose the function is rewritten to capture the variable by reference, `[&x]`, as follows. This will not work because the lambda expression will be executed later in the program, not anymore in the scope of the `multiplyBy2Lambda()` function at which point the reference to `x` is not valid anymore:

```
function<int(void)> multiplyBy2Lambda(int x)
{
    return [&x]{ return 2*x; }; // BUG!
}
```

Lambda Expressions as Parameters

You can write your own functions that accept lambda expressions as parameters. This can, for example, be used to implement callbacks. The following code implements a `testCallback()` function that accepts a `vector` of integers and a callback function. The implementation iterates over all the elements in the given `vector` and calls the callback function for each element. The callback function accepts the current element from the `vector` as an `int` argument and returns a Boolean. If the callback returns `false`, the iteration is stopped.

```
void testCallback(const vector<int>& vec,
                  const function<bool(int)>& callback)
{
    for (const auto& i : vec) {
        // Call callback. If it returns false, stop iteration.
        if (!callback(i)) {
            break;
        }
        // Callback did not stop iteration, so print value
        cout << i << " ";
    }
    cout << endl;
}
```

The `testCallback()` function can be tested as follows. First, a `vector` with 10 elements is created. Then the `testCallback()` function is called with a small lambda expression as a callback function. This lambda expression returns `true` for values that are less than 6.

```
vector<int> vec{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
testCallback(vec, [](int i){return i < 6;});
```

The output of this example is:

```
1 2 3 4 5
```

Examples with STL Algorithms

This section demonstrates lambda expressions with two STL algorithms. More examples follow later in this chapter.

count_if

The following example uses the `count_if()` algorithm to count the number of elements in the given vector that satisfy a certain condition. The condition is given in the form of a lambda expression, which captures the `value` variable from its enclosing scope by value.

```
vector<int> vec{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int value = 3;
int cnt = count_if(cbegin(vec), cend(vec),
                    [value](int i){ return i > value; });
cout << "Found " << cnt << " values > " << value << endl;
```

The output is as follows:

```
Found 6 values > 3
```

The example can be extended to demonstrate capturing variables by reference. The following lambda expression counts the number of times it is called by incrementing a variable in the enclosing scope that is captured by reference:

```
vector<int> vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int value = 3;
int cntLambdaCalled = 0;
int cnt = count_if(cbegin(vec), cend(vec),
                    [value, &cntLambdaCalled](int i){ ++cntLambdaCalled; return i > value; });
cout << "The lambda expression was called " << cntLambdaCalled
    << " times." << endl;
cout << "Found " << cnt << " values > " << value << endl;
```

The output is as follows:

```
The lambda expression was called 9 times.
Found 6 values > 3
```

generate

The `generate()` algorithm requires an iterator range and replaces the values in that range with the values returned from the function given as a third argument. The following example uses the `generate()` algorithm together with a lambda expression to put the numbers 2, 4, 8, 16, and so on in the vector:

```
vector<int> vec(10);
int value = 1;
generate(begin(vec), end(vec), [&value]{ value *= 2; return value; });
for (const auto& i : vec) {
    cout << i << " ";
}
```

The output is as follows:

```
2 4 8 16 32 64 128 256 512 1024
```

FUNCTION OBJECTS

You can overload the function call operator in a class such that objects of the class can be used in place of function pointers. These objects are called *function objects*, or just *functors*.

Many of the STL algorithms, such as `find_if()` and a version of `accumulate()`, require a function pointer as one of the parameters. When you use these functions, you can pass a functor instead of a lambda expression or function pointer. C++ provides several predefined functor classes, defined in the `<functional>` header file, that perform the most commonly used callback operations.

Functor classes often consist of simple one-line expressions. The clumsiness of having to create a function or functor class, give it a name that does not conflict with other names, and then use this name is considerable overhead for what is fundamentally a simple concept. In these cases, using anonymous (unnamed) functions represented by lambda expressions is a big convenience. Their syntax is easier and can make your code easier to understand. They are discussed in the previous sections. However, this section explains functors and how to use the predefined functor classes because you will likely encounter them at some point.

NOTE *It is recommended to use lambda expressions, if possible, instead of function objects because lambdas are easier to use and easier to understand.*

Arithmetic Function Objects

C++ provides functor class templates for the five binary arithmetic operators: `plus`, `minus`, `multiplies`, `divides`, and `modulus`. Additionally, unary `negate` is supplied. These classes are templated on the type of the operands and are wrappers for the actual operators. They take one or two parameters of the template type, perform the operation, and return the result. Here is an example using the `plus` class template:

```
plus<int> myPlus;
int res = myPlus(4, 5);
cout << res << endl;
```

This example is silly, because there's no reason to use the `plus` class template when you could just use `operator+` directly. The benefit of the arithmetic function objects is that you can pass them as callbacks to algorithms, which you cannot do directly with the arithmetic operators. For example, the implementation of the `geometricMean()` function earlier in this chapter used the `accumulate()` function with a function pointer to the `product()` callback to multiply two integers. You could rewrite it to use the predefined `multiplies` function object:

```
double geometricMean(const vector<int>& nums)
{
    double mult = accumulate(cbegin(nums), cend(nums), 1, multiplies<int>());
    return pow(mult, 1.0 / nums.size());
}
```

The expression `multiplies<int>()` creates a new object of the `multiplies` functor template class, instantiating it with the `int` type.

The other arithmetic function objects behave similarly.

WARNING *The arithmetic function objects are just wrappers around the arithmetic operators. If you use the function objects as callbacks in algorithms, make sure that the objects in your container implement the appropriate operation, such as `operator*` or `operator+`.*

C++14

Transparent Operator Functors

C++14 adds support for transparent operator functors, which allow you to omit the template type argument. For example, you can just specify `multiplies<>()` instead of `multiplies<int>()`:

```
double geometricMeanTransparent(const vector<int>& nums)
{
    double mult = accumulate(cbegin(nums), cend(nums), 1, multiplies<>());
    return pow(mult, 1.0 / nums.size());
}
```

A very important feature of these transparent operators is that they are heterogeneous. That is, they are not only more concise than the non-transparent functors, but they also have real functional advantages. For instance, the following code uses the transparent operator functor and uses 1.1, a `double`, as the initial value, while the `vector` contains integers. `accumulate()` will calculate the result as a `double` and `result` will be 6.6:

```
vector<int> nums{ 1, 2, 3 };
double result = accumulate(cbegin(nums), cend(nums), 1.1, multiplies<>());
```

If this code uses a non-transparent operator functor as follows, then `accumulate()` will calculate the result as an integer, and `result` will be 6. When you compile this code, the compiler will give you warnings about possible loss of data.

```
vector<int> nums{ 1, 2, 3 };
double result = accumulate(cbegin(nums), cend(nums), 1.1, multiplies<int>());
```

NOTE *It's recommended to always use the transparent operator functors.*

Comparison Function Objects

In addition to the arithmetic function object classes, the C++ language provides all the standard comparisons: `equal_to`, `not_equal_to`, `less`, `greater`, `less_equal`, and `greater_equal`. You've already seen `less` in Chapter 16 as the default comparison for elements in the `priority_queue` and the associative containers. Now you can learn how to change that criterion. Here's an example of a `priority_queue` using the default comparison operator: `less`.

```
priority_queue<int> myQueue;
myQueue.push(3);
myQueue.push(4);
myQueue.push(2);
myQueue.push(1);
while (!myQueue.empty()) {
```

```

    cout << myQueue.top() << " ";
    myQueue.pop();
}

```

The output from the program is:

```
4 3 2 1
```

As you can see, the elements of the queue are removed in descending order, according to the `less` comparison. You can change the comparison to `greater` by specifying it as the comparison template argument. The `priority_queue` template definition looks like this:

```
template <class T, class Container = vector<T>, class Compare = less<T> >;
```

Unfortunately, the `Compare` type parameter is last, which means that in order to specify it you must also specify the container. If you want to use a `priority_queue` that sorts the elements in ascending order using `greater`, then you need to change the definition of the `priority_queue` in the previous example to the following:

```
priority_queue<int, vector<int>, greater<>> myQueue;
```

The output now is as follows:

```
1 2 3 4
```

If your compiler does not yet support the transparent operator functors, then you need this definition:

```
priority_queue<int, vector<int>, greater<int>> myQueue;
```

Several algorithms that you learn about later in this chapter require comparison callbacks, for which the predefined comparators come in handy.

Logical Function Objects

C++ provides function object classes for the three logical operations: `logical_not` (`operator!`), `logical_and` (`operator&&`), and `logical_or` (`operator||`). These logical operations deal only with values `true` and `false`. Bitwise function objects are covered in the next section.

Logical functors can, for example, be used to implement an `allTrue()` function that checks if all the Boolean flags in a container are `true`:

```
bool allTrue(const vector<bool>& flags)
{
    return accumulate(begin(flags), end(flags), true, logical_and<>());
}
```

Similarly, the `logical_or` functor can be used to implement an `anyTrue()` function that returns `true` if there is at least one Boolean flag in a container `true`:

```
bool anyTrue(const vector<bool>& flags)
{
    return accumulate(begin(flags), end(flags), false, logical_or<>());
}
```

If your compiler does not yet support the transparent operator functors then you need to use `logical_and<bool>` and `logical_or<bool>` in the previous functions.

Bitwise Function Objects

C++ has function objects for all the bitwise operations: `bit_and` (`operator&`), `bit_or` (`operator|`), and `bit_xor` (`operator^`). C++14 adds `bit_not` (`operator~`). These bitwise functors can, for example, be used together with the `transform()` algorithm (discussed later in this chapter) to perform bitwise operations on all elements in a container.

Function Object Adapters

When you try to use the basic function objects provided by the standard, it often feels as if you're trying to put a square peg into a round hole. For example, you can't use the `less` function object with `find_if()` to find an element smaller than some value because `find_if()` passes only one argument to its callback each time instead of two. The *function adapters* attempt to rectify this problem and others. They provide a modicum of support for *functional composition*, or combining functions together to create the exact behavior you need.

Binders

Binders can be used to *bind* parameters of functions to certain values. For this you use `std::bind()`, defined in `<functional>`, which allows you to bind arguments of a function in a flexible way. You can bind function arguments to fixed values and you can even rearrange function arguments in a different order. It is best explained with an example.

Suppose you have a function called `func()` accepting two arguments:

```
void func(int num, const string& str)
{
    cout << "func(" << num << ", " << str << ")" << endl;
}
```

The following code demonstrates how you can use `bind()` to bind the second argument of `func()` to a fixed value, `myString`. The result is stored in `f1()`. The `auto` keyword is used to remove the need to specify the exact return type, which can become complicated. Arguments that are not bound to specific values should be specified as `_1`, `_2`, `_3`, and so on. These are defined in the `std::placeholders` namespace. In the definition of `f1()`, the `_1` specifies where the first argument to `f1()` needs to go when `func()` is called. After this, `f1()` can be called with just a single integer argument.

```
string myString = "abc";
auto f1 = bind(func, placeholders::_1, myString);
f1(16);
```

The output is:

```
func(16, abc)
```

`bind()` can also be used to rearrange the arguments, as shown in the following code. The `_2` specifies where the second argument to `f2()` needs to go when `func()` is called. In other words, the `f2()` binding means that the first argument to `f2()` will become the second argument to `func()`, and the second argument to `f2()` will become the first argument to `func()`.

```
auto f2 = bind(func, placeholders::_2, placeholders::_1);
f2("Test", 32);
```

The output is as follows:

```
func(32, Test)
```

The `<functional>` header defines the `std::ref()` and `std:: cref()` helper functions. These can be used to bind references or `const` references, respectively. For example, suppose you have the following function:

```
void increment(int& value) { ++value; }
```

If you call this function as follows, then the value of `index` becomes 1:

```
int index = 0;
increment(index);
```

If you use `bind()` to call it as follows, then the value of `index` is not incremented because a copy of `index` is made and a reference to this copy is bound to the first parameter of the `increment()` function:

```
int index = 0;
auto incr = bind(increment, index);
incr();
```

Using `std::ref()` to pass a proper reference correctly increments `index`:

```
int index = 0;
auto incr = bind(increment, ref(index));
incr();
```

There is a small issue with binding parameters in combination with overloaded functions. Suppose you have the following two overloaded functions called `overloaded()`. One accepts an integer and the other accepts a floating point number:

```
void overloaded(int num) {}
void overloaded(float f) {}
```

If you want to use `bind()` with these overloaded functions, you need to explicitly specify which of the two overloads you want to bind. The following will not compile:

```
auto f3 = bind(overloaded, placeholders::_1); // ERROR
```

If you want to bind the parameters of the overloaded function accepting a floating point argument, you need the following syntax:

```
auto f4 = bind((void*)(float))overloaded, placeholders::_1); // OK
```

Another example of `bind()` is to use the `find_if()` algorithm to find the first element in a sequence that is greater than or equal to 100. To solve this problem earlier in this chapter, a pointer to a `perfectScore()` function is passed to `find_if()`. This can be rewritten using the comparison functor `greater_equal` and `bind()`. The following code uses `bind()` to bind the second parameter of `greater_equal` to a fixed value of 100:

```
// Code for inputting scores into the vector omitted, similar as earlier.
auto endIter = end(myVector);
auto it = find_if(begin(myVector), endIter,
                  bind(greater_equal<>(), placeholders::_1, 100));
if (it == endIter) {
```

```

        cout << "No perfect scores" << endl;
    } else {
        cout << "Found a \"perfect\" score of " << *it << endl;
    }
}

```

WARNING Before C++11 there was `bind2nd()` and `bind1st()`. Both have been deprecated by C++11. Use lambda expressions or `bind()` instead.

Negators

The *negators* are functions similar to the binders but they complement the result of a predicate. For example, if you want to find the first element in a sequence of test scores less than 100, you could apply the `not1()` negator adapter to the result of `perfectScore()` like this:

```

// Code for inputting scores into the vector omitted, similar as earlier.
auto endIter = end(myVector);
function<bool(int)> f = perfectScore;
auto it = find_if(begin(myVector), endIter, not1(f));
if (it == endIter) {
    cout << "All perfect scores" << endl;
} else {
    cout << "Found a \"less-than-perfect\" score of " << *it << endl;
}

```

Note that in this example you could have used the `find_if_not()` algorithm. The function `not1()` complements the result of every call to the predicate it takes as an argument. The “1” in `not1()` refers to the fact that its operand must be a unary function (one that takes a single argument). If its operand is a binary function (takes two arguments), you must use `not2()` instead.

As you can see, using functors and adapters can become complicated. My advice is to use lambda expressions instead of functors if possible. For example, the previous `find_if()` call using the `not1()` negator can be written more elegantly using a lambda expression:

```
auto it = find_if(begin(myVector), endIter, [](int i){ return i < 100; });
```

Calling Member Functions

If you have a container of objects, you sometimes want to pass a pointer to a class method as the callback to an algorithm. For example, you might want to find the first empty `string` in a `vector` of `strings` by calling `empty()` on each `string` in the sequence. However, if you just pass a pointer to `string::empty()` to `find_if()`, the algorithm has no way to know that it received a pointer to a method instead of a normal function pointer or functor. The code to call a method pointer is different from that to call a normal function pointer, because the former must be called in the context of an object.

C++ provides a conversion function called `mem_fn()` that you can call on a method pointer before passing it to an algorithm. The following example demonstrates this. Note that you have to specify the method pointer as `&string::empty`. The `&string::` part is not optional.

```

void findEmptyString(const vector<string>& strings)
{
    auto endIter = end(strings);
    auto it = find_if(begin(strings), endIter, mem_fn(&string::empty));
    if (it == endIter) {
        cout << "No empty strings!" << endl;
    } else {
        cout << "Empty string at position: "
            << static_cast<int>(it - begin(strings)) << endl;
    }
}

```

`mem_fn()` generates a function object that serves as the callback for `find_if()`. Each time it is called back, it calls the `empty()` method on its argument.

`mem_fn()` works exactly the same when you have a container of pointers to objects instead of objects themselves. For example:

```

void findEmptyString(const vector<string*>& strings)
{
    auto endIter = end(strings);
    auto it = find_if(begin(strings), endIter, mem_fn(&string::empty));
    // Remainder of function omitted because it is the same as earlier
}

```

`mem_fn()` is not the most intuitive way to implement the `findEmptyString()` function. Using lambda expressions, it can be implemented in a much more readable and elegant way. Here is the implementation using a lambda expression working on a container of objects:

```

void findEmptyString(const vector<string>& strings)
{
    auto endIter = end(strings);
    auto it = find_if(begin(strings), endIter,
        [] (const string& str){ return str.empty(); });
    // Remainder of function omitted because it is the same as earlier
}

```

Similarly, the following uses a lambda expression working on a container of pointers to objects:

```

void findEmptyString(const vector<string*>& strings)
{
    auto endIter = end(strings);
    auto it = find_if(begin(strings), endIter,
        [] (const string* str){ return str->empty(); });
    // Remainder of function omitted because it is the same as earlier
}

```

Writing Your Own Function Objects

You can write your own function objects to perform more specific tasks than those provided by the predefined functors and if you need to do something more complex than suitable for lambda expressions. If you want to be able to use the function adapters with these custom functors, you must supply certain `typedefs`. The easiest way to do that is to derive your function object classes from either `unary_function` or `binary_function`, depending on whether they take one or two

arguments. These two classes, defined in `<functional>`, are templated on the parameter and return types of the “function” they provide. For example:

```
class myIsDigit : public unary_function<char, bool>
{
public:
    bool operator() (char c) const { return ::isdigit(c) != 0; }
};
bool isNumber(const string& str)
{
    auto endIter = end(str);
    auto it = find_if(begin(str), endIter, not1(myIsDigit()));
    return (it == endIter);
}
```

Note that the overloaded function call operator of the `myIsDigit` class must be `const` in order to pass objects of it to `find_if()`.

WARNING *The algorithms are allowed to make multiple copies of function object predicates and call different ones for different elements. The function call operator needs to be const; thus, you cannot write functors such that they count on any internal state to the object being consistent between calls.*

Before C++11, a class defined locally in the scope of a function could not be used as a template argument. This limitation has been removed. The following example demonstrates this:

```
bool isNumber(const string& str)
{
    class myIsDigit : public unary_function<char, bool>
    {
public:
    bool operator() (char c) const { return ::isdigit(c) != 0; }
    };
    auto endIter = end(str);
    auto it = find_if(begin(str), endIter, not1(myIsDigit()));
    return (it == endIter);
}
```

NOTE *As you can see from previous examples, lambda expressions allow you to write more readable and more elegant code. I recommend you use simple lambda expressions instead of function objects, and use function objects only when they need to do more complicated things.*

ALGORITHM DETAILS

Chapter 15 lists all available STL algorithms, divided into different categories. Most of the algorithms are defined in the `<algorithm>` header file, but a few are located in `<numeric>` and in `<utility>`. They are all in the `std` namespace. This chapter cannot discuss all available algorithms,

so it picks a number of categories and provides examples of those. Once you know how to use these, you should have no problems with the other algorithms. Consult a Standard Library Reference — for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/> — for a full reference of *all* the algorithms.

Iterators

First, a few more words on iterators. There are five types of iterators: input, output, forward, bidirectional, and random-access. These are described in Chapter 16. There is no formal class hierarchy of these iterators, because the implementations for each container are not part of the standard hierarchy. However, one can deduce a hierarchy based on the functionality they are required to provide. Specifically, every random access iterator is also bidirectional, every bidirectional iterator is also forward, and every forward iterator is also input and output. Figure 17-1 shows such hierarchy. Dotted lines are used because the figure is not a real class hierarchy.

The standard way for the algorithms to specify what kind of iterators they need is to use the following names for the iterator template arguments: `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and `RandomAccessIterator`. These names are just names: They don't provide binding type checking. Therefore, you could, for example, try to call an algorithm expecting a `RandomAccessIterator` by passing a bidirectional iterator. The template doesn't do type checking, so it would allow this instantiation. However, the code in the function that uses the random-access iterator capabilities would fail to compile on the bidirectional iterator. Thus, the requirement is enforced, just not where you would expect. The error message can therefore be somewhat confusing. For example, attempting to use the generic `sort()` algorithm, which requires a random-access iterator, on a `list`, which provides only a bidirectional iterator, gives a cryptic error of 32 lines in Visual C++ 2013, of which the first two lines are:

```
...\\vc\\include\\algorithm(3157): error C2784: 'unknown-type std::operator
-(std::move_iterator<_RanIt> &,const std::move_iterator<_RanIt2> &)' : could not
deduce template argument for 'std::move_iterator<_RanIt> &' from 'std::_List_
iterator<std::_List_val<std::_List_simple_types<int>>>'
```

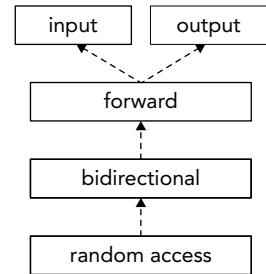


FIGURE 17-1

Non-Modifying Sequence Algorithms

The non-modifying sequence algorithms include functions for searching elements in a range, comparing two ranges to each other, and includes a number of utility algorithms.

Search Algorithms

You've already seen two examples of using search algorithms: `find()` and `find_if()`. The STL provides several other variations of the basic `find()` algorithm that work on sequences of elements. The section "Search Algorithms" in Chapter 15 describes the different search algorithms that are available, including their complexity.

All the algorithms use default comparisons of `operator==` or `operator<`, but also provide overloaded versions that allow you to specify a comparison callback.

Here are examples of some of the search algorithms:

```
// The list of elements to be searched
vector<int> myVector = { 5, 6, 9, 8, 8, 3 };
auto beginIter = cbegin(myVector);
auto endIter = cend(myVector);

// Find the first element that does not satisfy the given lambda expression
auto it = find_if_not(beginIter, endIter, [](int i){ return i < 8; });
if (it != endIter) {
    cout << "First element not < 8 is " << *it << endl;
}

// Find the min and max elements in the vector
auto res = minmax_element(beginIter, endIter);
cout << "min is " << *(res.first) << " and max is " << *(res.second) << endl;

// Find the min and max elements in the vector
it = min_element(beginIter, endIter);
auto it2 = max_element(beginIter, endIter);
cout << "min is " << *it << " and max is " << *it2 << endl;

// Find the first pair of matching consecutive elements
it = adjacent_find(beginIter, endIter);
if (it != endIter) {
    cout << "Found two consecutive equal elements with value " << *it << endl;
}

// Find the first of two values
vector<int> targets = { 8, 9 };
it = find_first_of(beginIter, endIter, cbegin(targets), cend(targets));
if (it != endIter) {
    cout << "Found one of 8 or 9: " << *it << endl;
}

// Find the first subsequence
vector<int> sub = { 8, 3 };
it = search(beginIter, endIter, cbegin(sub), cend(sub));
if (it != endIter) {
    cout << "Found subsequence {8,3}" << endl;
} else {
    cout << "Unable to find subsequence {8,3}" << endl;
}

// Find the last subsequence (which is the same as the first in this example)
it2 = find_end(beginIter, endIter, cbegin(sub), cend(sub));
if (it != it2) {
    cout << "Error: search and find_end found different subsequences "
        << "even though there is only one match." << endl;
}

// Find the first subsequence of two consecutive 8s
```

```

it = search_n(beginIter, endIter, 2, 8);
if (it != endIter) {
    cout << "Found two consecutive 8s" << endl;
} else {
    cout << "Unable to find two consecutive 8s" << endl;
}

```

Here is the output:

```

First element not < 8 is 9
min is 3 and max is 9
min is 3 and max is 9
Found two consecutive equal elements with value 8
Found one of 8 or 9: 9
Found subsequence {8,3}
Found two consecutive 8s

```

NOTE Remember that some of the containers have equivalent methods. If that's the case, it's recommended to use the methods instead of the generic algorithms, because the methods are more efficient.

Comparison Algorithms

You can compare entire ranges of elements in three different ways: `equal()`, `mismatch()`, and `lexicographical_compare()`. These algorithms have the advantage that you can compare ranges in different containers. For example, you can compare the contents of a `vector` with the contents of a `list`. In general, these work best with sequential containers. They work by comparing the values in corresponding positions of the two collections to each other.

- `equal()` returns `true` if all corresponding elements are equal. It requires both containers to have the same number of elements.
- `mismatch()` returns iterators, one iterator for each of the collections, to indicate where in the range the corresponding elements mismatched.
- `lexicographical_compare()` deals with the situation where the two ranges may contain different numbers of elements. It returns `true` if the first unequal element in the first range is less than its corresponding element in the second range, or, if the first range has fewer elements than the second and all elements in the first range are equal to their corresponding initial subsequence in the second set. “`lexicographical_compare`” gets its name because it resembles the rules for comparing strings, but extends this set of rules to deal with objects of any type.

NOTE If you want to compare the elements of two containers of the same type, you can use `operator==` or `operator<` instead of `equal()` or `lexicographical_compare()`. The algorithms are useful primarily for comparing sequences of elements from different container types.

Here are some examples of these algorithms:

```
// Function template to populate a container of ints.
// The container must support push_back().
template<typename Container>
void populateContainer(Container& cont)
{
    int num;
    while (true) {
        cout << "Enter a number (0 to quit): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        cont.push_back(num);
    }
}
int main()
{
    vector<int> myVector;
    list<int> myList;
    cout << "Populate the vector:" << endl;
    populateContainer(myVector);
    cout << "Populate the list:" << endl;
    populateContainer(myList);

    // compare the two containers
    if (myList.size() == myVector.size() &&
        equal(cbegin(myVector), cend(myVector), cbegin(myList))) {
        cout << "The two containers have equal elements" << endl;
    } else {
        if (myList.size() < myVector.size()) {
            cout << "Sorry, the list is not long enough." << endl;
            return 1;
        } else {
            // If the containers were not equal, find out why not
            auto miss = mismatch(cbegin(myVector), cend(myVector),
                                  cbegin(myList));
            cout << "The following initial elements are the same in "
                << "the vector and the list:" << endl;
            for (auto i = cbegin(myVector); i != miss.first; ++i) {
                cout << *i << '\t';
            }
            cout << endl;
        }
    }
}

// Now order them.
if (lexicographical_compare(cbegin(myVector), cend(myVector),
                           cbegin(myList), cend(myList))) {
    cout << "The vector is lexicographically first." << endl;
} else {
    cout << "The list is lexicographically first." << endl;
}
return 0;
}
```

Here is a sample run of the program:

```
Populate the vector:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 0
Populate the list:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 9
Enter a number (0 to quit): 8
Enter a number (0 to quit): 0
The following initial elements are the same in the vector and the list:
5      6
The vector is lexicographically first.
```



C++14 adds a version of `equal()` and `mismatch()` accepting four iterators: begin and end iterators for the first container, and begin and end iterators for the second container. Using these versions of `equal()` and `mismatch()` the tests in the previous example to check whether the size of `myList` is equal to or less than the size of `myVector` are not needed anymore:

```
if (equal(cbegin(myVector), cend(myVector), cbegin(myList), cend(myList))) {
    cout << "The two containers have equal elements" << endl;
} else {
    // If the containers were not equal, find out why not
    auto miss = mismatch(cbegin(myVector), cend(myVector),
        cbegin(myList), cend(myList));
    cout << "The following initial elements are the same in "
        << "the vector and the list:" << endl;
    for (auto i = cbegin(myVector); i != miss.first; ++i) {
        cout << *i << '\t';
    }
    cout << endl;
}
```

Utility Algorithms

The non-modifying utility algorithms are: `all_of()`, `any_of()`, `none_of()`, `count()`, and `count_if()`. Here are examples of the first three algorithms. An example of `count_if()` is given earlier in this chapter.

```
// all_of()
vector<int> vec2 = { 1, 1, 1, 1 };
if (all_of(cbegin(vec2), cend(vec2), [](int i){ return i == 1; })) {
    cout << "All elements are == 1" << endl;
} else {
    cout << "Not all elements are == 1" << endl;
}

// any_of()
vector<int> vec3 = { 0, 0, 1, 0 };
if (any_of(cbegin(vec3), cend(vec3), [](int i){ return i == 1; })) {
    cout << "At least one element == 1" << endl;
} else {
```

```

        cout << "No elements are == 1" << endl;
    }

// none_of()
vector<int> vec4 = { 0, 0, 0, 0 };
if (none_of(cbegin(vec4), cend(vec4), [](int i){ return i == 1; })) {
    cout << "All elements are != 1" << endl;
} else {
    cout << "Some elements are == 1" << endl;
}

```

The output is as follows:

```

All elements are == 1
At least one element == 1
All elements are != 1

```

Modifying Sequence Algorithms

The STL provides a variety of *modifying sequence algorithms* that perform tasks such as copying elements from one range to another, removing elements, or reversing the order of elements in a range.

The modifying algorithms all have the concept of *source* and *destination* ranges. The elements are read from the source range and added to or modified in the destination range. The source and destination ranges can often be the same, in which case the algorithm is said to operate *in place*.

WARNING Ranges from `maps` and `multimaps` cannot be used as destinations of modifying algorithms. These algorithms overwrite entire elements, which in a `map` consist of key/value pairs. However, `maps` and `multimaps` mark the key `const`, so it cannot be assigned to. The same holds for `set` and `multiset`. Your alternative is to use an `insert` iterator, described in Chapter 20.

The section “Modifying Sequence Algorithms” in Chapter 15 lists all available modifying algorithms with a description. This section provides code examples for a number of those algorithms. If you understand how to use the algorithms explained in this section, you should have no problems using the other algorithms for which no examples are given.

transform

The `transform()` algorithm applies a callback to each element in a range and expects the callback to generate a new element, which it stores in the destination range specified. The source and destination ranges can be the same if you want `transform()` to replace each element in a range with the result from the call to the callback. The parameters are a begin and end iterator of the source sequence, a begin iterator of the destination sequence, and the callback. For example, you could add 100 to each element in a `vector` like this:

```

vector<int> myVector;
populateContainer(myVector);
cout << "The vector contents are:" << endl;

```

```

for (const auto& i : myVector) { cout << i << " "; }
cout << endl;

transform(begin(myVector), end(myVector), begin(myVector),
[](int i){ return i + 100;});

cout << "The vector contents are:" << endl;
for (const auto& i : myVector) { cout << i << " "; }

```

Another form of `transform()` calls a binary function on pairs of elements in a range. It requires a begin and end iterator of the first range, a begin iterator of the second range, and a begin iterator of the destination range. The following example creates two vectors and uses `transform()` to calculate the sum of pairs of elements and store the result back in the first vector:

```

vector<int> vec1;
cout << "Vector1:" << endl;
populateContainer(vec1);
cout << "Vector2:" << endl;
vector<int> vec2;
populateContainer(vec2);
if (vec2.size() < vec1.size())
{
    cout << "Vector2 should be at least the same size as vector1." << endl;
    return 1;
}
// Create a lambda to print a container
auto printContainer = [] (const auto& container) {
    for (auto& i : container) { cout << i << " "; }
    cout << endl;
};
cout << "Vector1: "; printContainer(vec1);
cout << "Vector2: "; printContainer(vec2);

transform(begin(vec1), end(vec1), begin(vec2), begin(vec1),
[](int a, int b){return a + b;});

cout << "Vector1: "; printContainer(vec1);
cout << "Vector2: "; printContainer(vec2);

```

The output could look as follows:

```

Vector1:
Enter a number (0 to quit): 1
Enter a number (0 to quit): 2
Enter a number (0 to quit): 0
Vector2:
Enter a number (0 to quit): 11
Enter a number (0 to quit): 22
Enter a number (0 to quit): 33
Enter a number (0 to quit): 0
Vector1: 1 2
Vector2: 11 22 33
Vector1: 12 24
Vector2: 11 22 33

```

NOTE `transform()` and the other modifying algorithms often return an iterator referring to the past-the-end value of the destination range. The examples in this book usually ignore that return value.

copy

The `copy()` algorithm allows you to copy elements from one range to another, starting with the first element and proceeding to the last element in the range. The source and destination ranges must be different, but they can overlap. Note that `copy()` doesn't `insert` elements into the destination range. It just `overwrites` whatever elements were there already. Thus, you can't use `copy()` directly to insert elements into a container, only to overwrite elements that were previously in a container.

NOTE Chapter 20 describes how to use iterator adapters to insert elements into a container or stream with `copy()`.

Here is a simple example of `copy()` that uses the `resize()` method on a `vector` to ensure that there is enough space in the destination container. It copies all elements from `vec1` to `vec2`:

```
vector<int> vec1, vec2;
populateContainer(vec1);
vec2.resize(vec1.size());
copy(cbegin(vec1), cend(vec1), begin(vec2));
for (const auto& i : vec2) { cout << i << " ";
```

There is also a `copy_backward()` algorithm, which copies the elements from the source backward to the destination. In other words, it starts with the last element of the source and puts it in the last position in the destination range and moves backward after each copy. The preceding example can be modified to use `copy_backward()` instead of `copy()`, as follows. Note that you need to specify `end(vec2)` as a third argument instead of `begin(vec2)`:

```
copy_backward(cbegin(vec1), cend(vec1), end(vec2));
```

This results in exactly the same output.

`copy_if()` works by having an input range specified by two iterators, an output destination specified by one iterator, and a predicate (function or lambda expression). The function or lambda expression is executed for each element that is a candidate to be copied. If the returned value is `true`, the element is copied and the destination iterator is incremented; if the value is `false` the element is not copied and the destination iterator is not incremented. Thus, the destination may hold fewer elements than the source range. For some containers, because they must have already created space to hold the maximum possible number of elements (remember, `copy` does not create or extend containers, merely replaces the existing elements), it might be desirable to remove the space "beyond" where the last element was copied to. To facilitate this, `copy_if()` returns an iterator to the one-past-the-last-copied element in the destination range. This can be used to determine how many elements should be removed from the destination container. The following example demonstrates this by copying only the even numbers to `vec2`:

```

vector<int> vec1, vec2;
populateContainer(vec1);
vec2.resize(vec1.size());
auto endIterator = copy_if(cbegin(vec1), cend(vec1),
    begin(vec2), [] (int i){ return i % 2 == 0; });
vec2.erase(endIterator, end(vec2));
for (const auto& i : vec2) { cout << i << " ";}

```

`copy_n()` copies *n* elements from the source to the destination. The first parameter of `copy_n()` is the start iterator. The second parameter of `copy_n()` is an integer specifying the number of elements to copy and the third parameter is the destination iterator. The `copy_n()` algorithm does not perform any bounds checking, so you must make sure that the start iterator, incremented by the number of elements to copy, does not exceed the `end()` of the collection or your program will have undefined behavior. The following is an example:

```

vector<int> vec1, vec2;
populateContainer(vec1);
size_t cnt = 0;
cout << "Enter number of elements you want to copy: ";
cin >> cnt;
cnt = min(cnt, vec1.size());
vec2.resize(cnt);
copy_n(cbegin(vec1), cnt, begin(vec2));
for (const auto& i : vec2) { cout << i << " ";}

```

move

There are two move-related algorithms: `move()` and `move_backward()`. They both use move semantics discussed in Chapter 10. You have to provide a move assignment operator in your element classes if you want to use these algorithms on containers with elements of your own types, as demonstrated in the following example. The `main()` function creates a vector with three `MyClass` objects and then moves those elements from `vecSrc` to `vecDst`. Note that the code includes two different uses of `move()`. The `move()` function accepting a single argument converts an lvalue into an rvalue, while `move()` accepting three arguments is the STL `move()` algorithm to move elements between containers. Consult Chapter 10 for details on implementing move assignment operators and the use of the single parameter version of `std::move()`.

```

class MyClass
{
public:
    MyClass() = default;
    MyClass(const MyClass& src) = default;
    MyClass(const string& str) : mStr(str) {}
    // Move assignment operator
    MyClass& operator=(MyClass&& rhs) noexcept {
        if (this == &rhs)
            return *this;
        mStr = std::move(rhs.mStr);
        cout << "Move operator= (mStr=" << mStr << ")" << endl;
        return *this;
    }
    string getString() const {return mStr;}
private:

```

```

        string mStr;
    };
    int main()
    {
        vector<MyClass> vecSrc {MyClass("a"), MyClass("b"), MyClass("c")};
        vector<MyClass> vecDst(vecSrc.size());
        move(begin(vecSrc), end(vecSrc), begin(vecDst));
        for (const auto& c : vecDst) { cout << c.getString() << " ";}
        return 0;
    }
}

```

The output is as follows:

```

Move operator= (mStr=a)
Move operator= (mStr=b)
Move operator= (mStr=c)
a b c

```

NOTE Chapter 10 explains that source objects in a move operation are reset because the target object takes ownership of the resources of the source object. For the previous example, this means that you should not use the elements from vecSrc anymore after the move operation.

`move_backward()` uses the same move mechanism as `move()` but it moves the elements starting from the last to the first element.

replace

The `replace()` and `replace_if()` algorithms replace elements in a range matching a value or predicate, respectively, with a new value. Take `replace_if()` as an example. Its first and second parameters specify the range of elements in your container. The third parameter is a function or lambda expression that returns `true` or `false`. If it returns `true`, the value in the container is replaced with the value given as fourth parameter; if it returns `false`, it leaves the original value.

For example, you might want to replace all values less than a lower limit with the value of the lower limit and replace all values greater than an upper limit with the value of the upper limit. This is called “clamping” values to a range. In audio applications, this is known as “clipping.” For example, audio signals are often limited to integers in the range of -32K to +32K. The following example demonstrates clipping by first calling `replace_if()` to replace all values less than -32K with -32K and then calling it a second time to replace all values greater than 32K with 32K:

```

vector<int> vec;
populateContainer(vec);
int low = -32768;
int up = 32767;
replace_if(begin(vec), end(vec), [low](int i){ return i < low; }, low);
replace_if(begin(vec), end(vec), [up](int i){ return i > up; }, up);
for (const auto& i : vec) { cout << i << " ";}

```

There are also variants of `replace()` called `replace_copy()` and `replace_copy_if()` that copy the results to a different destination range. They are similar to `copy()`, in that the destination range must already be large enough to hold the new elements.

remove

Suppose you have a range of elements and you want to remove elements matching a certain condition. The first solution that you might think of is to check the documentation to see if your container has an `erase()` method and then iterate over all the elements and call `erase()` for each element that matches the condition. The `vector` is an example of a container that has such an `erase()` method. However, if applied to the `vector` container, this solution is very inefficient as it will cause a lot of memory operations to keep the `vector` contiguous in memory, resulting in a quadratic complexity (see Chapter 4). This solution is also error-prone, because you need to be careful that you keep your iterators valid after a call to `erase()`. The correct solution for this problem is the so-called *remove-erase-idiom*, which runs in linear time and is explained in this section.

Algorithms have access only to the iterator abstraction, not to the container. Thus the `remove` algorithms cannot really remove them from the underlying container. Instead, the algorithms work by replacing the elements that match the given value or predicate with the next element that does not match the given value or predicate. The result is that the range becomes partitioned into two sets: the elements to be kept and elements to be removed. An iterator is returned that points to the first element in the range of elements to be removed. If you want to actually erase these elements from the container, you must use the `remove()` algorithm, then call `erase()` on the container to erase all the elements from the returned iterator up to the end of the range. This is the *remove-erase-idiom*. Here is an example of a function that removes empty strings from a `vector` of strings:

```
void removeEmptyStrings(vector<string>& strings)
{
    auto it = remove_if(begin(strings), end(strings),
        [] (const string& str){ return str.empty(); });
    // Erase the removed elements.
    strings.erase(it, end(strings));
}
int main()
{
    vector<string> myVector = {"", "one", "", "two", "three", "four"};
    for (auto& str : myVector) { cout << "\"" << str << "\" ";}
    cout << endl;
    removeEmptyStrings(myVector);
    for (auto& str : myVector) { cout << "\"" << str << "\" ";}
    cout << endl;
    return 0;
}
```

The output is as follows:

```
"" "one" "" "two" "three" "four"
"one" "two" "three" "four"
```

The `remove_copy()` and `remove_copy_if()` variations of `remove()` do not change the source range. Instead they copy all kept elements to a different destination range. They are similar to `copy()`, in that the destination range must already be large enough to hold the new elements.

NOTE *The `remove()` family of functions are stable in that they maintain the order of elements remaining in the container even while moving the retained elements toward the beginning.*

unique

The `unique()` algorithm is a special case of `remove()` that removes all duplicate contiguous elements. The `list` container provides its own `unique()` method that implements the same semantics. You should generally use `unique()` on sorted sequences, but nothing prevents you from running it on unsorted sequences.

The basic form of `unique()` runs in place, but there is also a version of the algorithm called `unique_copy()` that copies its results to a new destination range.

Chapter 16 shows an example of the `list::unique()` algorithm, so an example of the general form here is omitted.

reverse

The `reverse()` algorithm reverses the order of the elements in a range. The first element in the range is swapped with the last, the second with the second-to-last, and so on.

The basic form of `reverse()` runs in place and requires two arguments: a start and end iterator for the range. There is also a version of the algorithm called `reverse_copy()` that copies its results to a new destination range and requires three arguments: a start and end iterator for the source range and a start iterator for the destination range. The destination range must already be large enough to hold the new elements.

shuffle

`shuffle()` rearranges the elements of a range in a random order with a linear complexity. It's useful for implementing tasks like shuffling a deck of cards. `shuffle()` requires a start and end iterator for the range that you want to shuffle and a uniform random number generator object that specifies how the random numbers should be generated. Random number generators are discussed in detail in Chapter 19.

Operational Algorithms

There is only one algorithm in this category: `for_each()`. It executes a callback on each element of the range. You can use it with simple function callbacks or lambda expressions for things like printing every element in a container. `for_each()` is mentioned here because you might encounter it in existing code; however, it's often easier and more readable to use a simple range-based `for` loop instead of `for_each()`.

Following is an example using a lambda expression, printing the elements from a map:

```
map<int, int> myMap = { { 4, 40 }, { 5, 50 }, { 6, 60 } };
for_each(cbegin(myMap), cend(myMap), [] (const pair<int, int>& p)
    { cout << p.first << "->" << p.second << endl;});
```

The output is as follows:

```
4->40
5->50
6->60
```

In C++14 you can use a generic lambda expression using the `auto` keyword:

```
for_each(cbegin(myMap), cend(myMap), [] (const auto& p)
    { cout << p.first << "->" << p.second << endl;});
```

Without lambda expressions, you have to write a separate function and pass a pointer to it to `for_each()`.

The following example shows how to use the `for_each()` algorithm and a lambda expression to calculate the sum and the product of a range of elements at the same time. Note that the lambda expression explicitly captures only those variables it needs. It captures them by reference; otherwise, changes made to `sum` and `prod` in the lambda expression would not be visible outside the lambda.

```
vector<int> myVector;
populateContainer(myVector);
int sum = 0;
int prod = 1;
for_each(cbegin(myVector), cend(myVector),
    [&sum, &prod] (int i) {
        sum += i;
        prod *= i;
    });
cout << "The sum is " << sum << endl;
cout << "The product is " << prod << endl;
```

This example can also be written with a functor in which you accumulate information that you can retrieve after `for_each()` has finished processing each element. For example, you could calculate both the sum and product of elements in one pass by writing a functor `SumAndProd` that tracks both at the same time:

```
class SumAndProd : public unary_function<int, void>
{
public:
    SumAndProd() : mSum(0), mProd(1) {}
    void operator()(int elem);
    int getSum() const { return mSum; }
    int getProduct() const { return mProd; }
private:
    int mSum;
    int mProd;
};
void SumAndProd::operator()(int elem)
{
    mSum += elem;
    mProd *= elem;
```

```

    }
int main()
{
    vector<int> myVector;
    populateContainer(myVector);
    SumAndProd func;
    func = for_each(cbegin(myVector), cend(myVector), func);
    cout << "The sum is " << func.getSum() << endl;
    cout << "The product is " << func.getProduct() << endl;
    return 0;
}

```

You might be tempted to ignore the return value of `for_each()`, yet still try to read information from `func` after the call. However, that doesn't work because the functor is moved into the `for_each()`, and at the end, it is moved back out of the `for_each()`. You must capture the return value in order to ensure correct behavior.

A final point about `for_each()` is that your lambda or callback is allowed to take its argument by reference and modify it. That has the effect of changing values in the actual iterator range. The voter registration example later in this chapter shows a use of this capability.

Partition Algorithms

`partition_copy()` copies elements from a source to two different destinations. The specific destination for each element is selected based on the result of a predicate, either `true` or `false`. The returned value of `partition_copy()` is a pair of iterators: one iterator referring to one-past-the-last-copied element in the first destination range, and one iterator referring to one-past-the-last-copied element in the second destination range. These returned iterators can be used in combination with `erase()` to remove excess elements from the two destination ranges, just as with the `copy_if()` example earlier. The following example asks the user to enter a number of integers, which are then *partitioned* into two destination vectors; one for the even numbers and one for the odd numbers:

```

vector<int> vec1, vecOdd, vecEven;
populateContainer(vec1);
vecOdd.resize(vec1.size());
vecEven.resize(vec1.size());

auto pairIters = partition_copy(cbegin(vec1), cend(vec1),
    begin(vecEven), begin(vecOdd),
    [] (int i){ return i % 2 == 0; });

vecEven.erase(pairIters.first, end(vecEven));
vecOdd.erase(pairIters.second, end(vecOdd));
cout << "Even numbers: ";
for (const auto& i : vecEven) { cout << i << " "; }
cout << endl << "Odd numbers: ";
for (const auto& i : vecOdd) { cout << i << " "; }

```

The output can be as follows:

```

Enter a number (0 to quit): 11
Enter a number (0 to quit): 22
Enter a number (0 to quit): 33
Enter a number (0 to quit): 44

```

```
Enter a number (0 to quit): 0
Even numbers: 22 44
Odd numbers: 11 33
```

The `partition()` algorithm sorts a sequence such that all elements for which a predicate returns `true` are before all elements for which it returns `false`, without preserving the original order of the elements within each partition. The following example demonstrates how to partition a `vector` into all even numbers followed by all odd numbers.

```
vector<int> vec;
populateContainer(vec);

partition(begin(vec), end(vec), [](int i){ return i % 2 == 0; });

cout << "Partitioned result: ";
for (const auto& i : vec) { cout << i << " "; }
```

The output can be as follows:

```
Enter a number (0 to quit): 55
Enter a number (0 to quit): 44
Enter a number (0 to quit): 33
Enter a number (0 to quit): 22
Enter a number (0 to quit): 11
Enter a number (0 to quit): 0
Partitioned result: 22 44 33 55 11
```

A couple of other partition algorithms are available. See Chapter 15 for a list.

Sorting Algorithms

The STL provides several variations of sorting algorithms. A “sorting algorithm” reorders the contents of a container such that an ordering is maintained between sequential elements of the collection. Thus, it applies only to sequential collections. Sorting is not relevant to associative containers because they already maintain elements in a sorted order. Sorting is not relevant to the unordered associative containers either because they have no concept of ordering. Some containers, such as `list` and `forward_list`, provide their own sorting methods because these can be implemented more efficiently than a general sort mechanism. Consequently, the general sorting algorithms are most useful for `vectors`, `deques`, and `arrays`.

The `sort()` function sorts a range of elements in $O(N \log N)$ time in the general case. Following the application of `sort()` to a range, the elements in the range are in nondecreasing order (lowest to highest), according to `operator<`. If you don’t like that order, you can specify a different comparison callback such as `greater`.

A variant of `sort()`, called `stable_sort()`, maintains the relative order of equal elements in the range. However, because it needs to maintain relative order of equal elements in the range, it is less efficient than the `sort()` algorithm.

Here is an example:

```
vector<int> vec;
cout << "Enter values:" << endl;
populateContainer(vec);
sort(begin(vec), end(vec));
```

There is also `is_sorted()` and `is_sorted_until()`; `is_sorted()` returns `true` if the given range is sorted, while `is_sorted_until()` returns an iterator in the given range such that everything before this iterator is sorted.

Binary Search Algorithms

There are several search algorithms that work only on sequences that are sorted or that are at least partitioned on the element that is searched for. This could, for example, be achieved by applying `partition()`. These algorithms are: `binary_search()`, `lower_bound()`, `upper_bound()`, and `equal_range()`. Examples of sorted sequences are `vectors` whose contents are sorted, `map`, `multimap`, `set`, and `multiset`. The `lower_bound()`, `upper_bound()`, and `equal_range()` algorithms are similar to their method equivalents on the `map` and `set` containers. See Chapter 16 for an example on how to use them.

The `binary_search()` algorithm finds a matching element in logarithmic time instead of linear time. It requires a start and end iterator specifying the range, a value to search, and optionally a comparison callback. It returns `true` if the value is found in the specified range, `false` otherwise. The following example demonstrates this algorithm:

```
vector<int> vec;
cout << "Enter values:" << endl;
populateContainer(vec);
// Sort the container
sort(begin(vec), end(vec));
while (true) {
    int num;
    cout << "Enter a number to find (0 to quit): ";
    cin >> num;
    if (num == 0) {
        break;
    }
    if (binary_search(cbegin(vec), cend(vec), num)) {
        cout << "That number is in the vector." << endl;
    } else {
        cout << "That number is not in the vector." << endl;
    }
}
```

Set Algorithms

The set algorithms work on any sorted iterator range. The `includes()` algorithm implements standard subset determination, checking if all the elements of one sorted range are included in another sorted range, in any order.

The `set_union()`, `set_intersection()`, `set_difference()`, and `set_symmetric_difference()` algorithms implement the standard semantics of those operations. In set theory, the result of union is all the elements in either set. The result of intersection is all the elements, which are in both sets. The result of difference is all the elements in the first set but not the second. The result of symmetric difference is the “exclusive or” of sets: all the elements in one, but not both, sets.

WARNING Make sure that your result range is large enough to hold the result of the operations. For `set_union()` and `set_symmetric_difference()`, the result is at most the sum of the sizes of the two input ranges. For `set_intersection()`, the result is at most the minimum size of the two input ranges, and for `set_difference()` it's at most the size of the first range.

WARNING You can't use iterator ranges from associative containers, including sets, to store the results because they don't allow changes to their keys.

Here are examples of how to use these algorithms:

```
vector<int> vec1, vec2, result;
cout << "Enter elements for set 1:" << endl;
populateContainer(vec1);
cout << "Enter elements for set 2:" << endl;
populateContainer(vec2);
// set algorithms work on sorted ranges
sort(begin(vec1), end(vec1));
sort(begin(vec2), end(vec2));
cout << "Set 1: ";
for (const auto& i : vec1) { cout << i << " "; }
cout << endl;
cout << "Set 2: ";
for (const auto& i : vec2) { cout << i << " "; }
cout << endl;
if (includes(cbegin(vec1), cend(vec1), cbegin(vec2), cend(vec2))) {
    cout << "The second set is a subset of the first." << endl;
}
if (includes(cbegin(vec2), cend(vec2), cbegin(vec1), cend(vec1))) {
    cout << "The first set is a subset of the second" << endl;
}
result.resize(vec1.size() + vec2.size());
auto newEnd = set_union(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result));
cout << "The union is: ";
for_each(begin(result), newEnd, [](int i){ cout << i << " "; });
cout << endl;
newEnd = set_intersection(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result));
cout << "The intersection is: ";
for_each(begin(result), newEnd, [](int i){ cout << i << " "; });
cout << endl;
newEnd = set_difference(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result));
cout << "The difference between set 1 and set 2 is: ";
for_each(begin(result), newEnd, [](int i){ cout << i << " "; });
cout << endl;
```

```

newEnd = set_symmetric_difference(cbegin(vec1), cend(vec1),
                                 cbegin(vec2), cend(vec2), begin(result));
cout << "The symmetric difference is: ";
for_each(begin(result), newEnd, [](int i){ cout << i << " "; });
cout << endl;

```

Here is a sample run of the program:

```

Enter elements for set 1:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 8
Enter a number (0 to quit): 0
Enter elements for set 2:
Enter a number (0 to quit): 8
Enter a number (0 to quit): 9
Enter a number (0 to quit): 10
Enter a number (0 to quit): 0
Set 1: 5 6 7 8
Set 2: 8 9 10
The union is: 5 6 7 8 9 10
The intersection is: 8
The difference between set 1 and set 2 is: 5 6 7
The symmetric difference is: 5 6 7 9 10

```

The `merge()` function allows you to merge two sorted ranges together, while maintaining the sorted order. The result is a sorted range containing all the elements of the two source ranges. It works in linear time. The following parameters are required:

- start and end iterator of first source range
- start and end iterator of second source range
- start iterator of destination range
- optionally, a comparison callback

Without `merge()`, you could still achieve the same effect by concatenating the two ranges and applying `sort()` to the result, but that would be less efficient [$O(N \log N)$ instead of linear].

WARNING *Always ensure that you supply a big enough destination range to store the result of the merge!*

The following example demonstrates `merge()`.

```

vector<int> vectorOne, vectorTwo, vectorMerged;
cout << "Enter values for first vector:" << endl;
populateContainer(vectorOne);
cout << "Enter values for second vector:" << endl;
populateContainer(vectorTwo);
// Sort both containers

```

```

sort(begin(vectorOne), end(vectorOne));
sort(begin(vectorTwo), end(vectorTwo));
// Make sure the destination vector is large enough to hold the values
// from both source vectors.
vectorMerged.resize(vectorOne.size() + vectorTwo.size());

merge(cbegin(vectorOne), cend(vectorOne), cbegin(vectorTwo),
      cend(vectorTwo), begin(vectorMerged));

cout << "Merged vector: ";
for (const auto& i : vectorMerged) { cout << i << " "; }
cout << endl;

```

Minimum/Maximum Algorithms

The `min()` and `max()` algorithms compare two or more elements of any type using `operator<` or a user-supplied binary predicate, returning a `const` reference to the smallest or largest element, respectively. The `minmax()` algorithm returns a pair containing the minimum and maximum value of two or more elements. These algorithms do not take iterator parameters. There is also `min_element()`, `max_element()`, and `minmax_element()` that work on iterator ranges.

The following program gives some examples:

```

int x = 4, y = 5;
cout << "x is " << x << " and y is " << y << endl;
cout << "Max is " << max(x, y) << endl;
cout << "Min is " << min(x, y) << endl;

// Using max() and min() on more than two values
int x1 = 2, x2 = 9, x3 = 3, x4 = 12;
cout << "Max of 4 elements is " << max({ x1, x2, x3, x4 }) << endl;
cout << "Min of 4 elements is " << min({ x1, x2, x3, x4 }) << endl;

// Using minmax()
auto p2 = minmax({ x1, x2, x3, x4 });
cout << "Minmax of 4 elements is <" 
     << p2.first << "," << p2.second << ">" << endl;

// Using minmax_element()
vector<int> vec{ 11, 33, 22 };
auto result = minmax_element(cbegin(vec), cend(vec));
cout << "minmax_element() result: <" 
     << *result.first << "," << *result.second << ">" << endl;

```

Here is the program output:

```

x is 4 and y is 5
Max is 5
Min is 4
Max of 4 elements is 12
Min of 4 elements is 2
Minmax of 4 elements is <2,12>
minmax_element() result: <11,33>

```

NOTE Sometimes you might encounter non-standard macros to find the minimum and maximum. For example, the GNU C Library (glibc) has macros `MIN()` and `MAX()`, while the `Windows.h` header file defines `min()` and `max()` macros. Because these are macros, they potentially evaluate one of their arguments twice; whereas `std::min()` and `std::max()` evaluate each argument exactly once. Make sure you always use the C++ versions, `std::min()` and `std::max()`.

Numerical Processing Algorithms

You've already seen an example of one numerical processing algorithm: `accumulate()`. The following sections give examples of two more numerical algorithms.

inner_product

`inner_product()`, defined in `<numeric>`, calculates the inner product of two sequences. For example, the inner product in the following example is calculated as $(1*9) + (2*8) + (3*7) + (4*6)$:

```
vector<int> v1{ 1, 2, 3, 4 };
vector<int> v2{ 9, 8, 7, 6 };
cout << inner_product(cbegin(v1), cend(v1), cbegin(v2), 0) << endl;
```

The output is 70.

iota

The `iota()` algorithm, defined in the `<numeric>` header file, generates a sequence of values in the specified range starting with the specified value and applying `operator++` to generate each successive value. The following example shows how to use this algorithm on a `vector` of integers, but note that it works on any element type that implements `operator++`:

```
vector<int> vec(10);
iota(begin(vec), end(vec), 5);
for (auto& i : vec) { cout << i << " "; }
```

The output is as follows:

```
5 6 7 8 9 10 11 12 13 14
```

ALGORITHMS EXAMPLE: AUDITING VOTER REGISTRATIONS

Voter fraud can be a problem everywhere. People sometimes attempt to register and vote in two or more different voting districts. Additionally, some people, for example convicted felons, are ineligible to vote, but occasionally attempt to register and vote anyway. Using your newfound algorithm skills, you could write a simple voter registration auditing function that checks the voter rolls for certain anomalies.

The Voter Registration Audit Problem Statement

The voter registration audit function should audit the voters' information. Assume that voter registrations are stored by district in a `map` that maps district names to a `list` of voters. Your audit function should take this `map` and a `list` of convicted felons as parameters, and should remove all convicted felons from the `lists` of voters. Additionally, the function should find all voters who are registered in more than one district and should remove those names from all districts. Voters with duplicate registrations must have all their registrations removed, and therefore become ineligible to vote. For simplicity, assume that the `list` of voters is simply a `list` of `string` names. A real application would obviously require more data, such as address and party affiliation.

The `auditVoterRolls` Function

The `auditVoterRolls()` function works in three steps:

1. Find all the duplicate names in all the registration `lists` by making a call to `getDuplicates()`.
2. Combine the `set` of duplicates and the `list` of convicted felons.
3. Remove from every voter `list` all the names found in the combined `set` of duplicates and convicted felons. The approach taken here is to use `for_each()` to process each `list` in the `map`, applying a lambda expression to remove the offending names from each `list`.

The following type aliases are used in the code:

```
using VotersMap = map<string, list<string>>;
using DistrictPair = pair<const string, list<string>>;
```

Here's the implementation of `auditVoterRolls()`:

```
// Expects a map of string/list<string> pairs keyed on district names
// and containing lists of all the registered voters in those districts.
// Removes from each list any name on the convictedFelons list and
// any name that is found on any other list.
void auditVoterRolls(VotersMap& votersByDistrict,
                      const list<string>& convictedFelons)
{
    // get all the duplicate names
    set<string> toRemove = getDuplicates(votersByDistrict);
    // combine the duplicates and convicted felons -- we want
    // to remove names on both lists from all voter rolls
    toRemove.insert(cbegin(convictedFelons), cend(convictedFelons));
    // Now remove all the names we need to remove using
    // nested lambda expressions and the remove-erase-idiom
    for_each(begin(votersByDistrict), end(votersByDistrict),
              [&toRemove] (DistrictPair& district) {
        auto it = remove_if(begin(district.second),
                            end(district.second), [&toRemove] (const string& name) {
            return (toRemove.count(name) > 0);
        });
        district.second.erase(it, end(district.second));
    });
}
```

The `getDuplicates` Function

The `getDuplicates()` function must find any name that is on more than one voter registration list. There are several different approaches one could use to solve this problem. To demonstrate the `adjacent_find()` algorithm, this implementation combines the lists from each district into one big list and sorts it. At that point, any duplicate names between the different lists will be next to each other in the big list. `getDuplicates()` can then use the `adjacent_find()` algorithm on the big, sorted list to find all consecutive duplicates and store them in a set called `duplicates`. Here is the implementation:

```
// Returns a set of all names that appear in more than one list in
// the map.
set<string> getDuplicates(const VotersMap& votersByDistrict)
{
    // Collect all the names from all the lists into one big list
    list<string> allNames;
    for (auto& district : votersByDistrict) {
        allNames.insert(end(allNames), begin(district.second),
                        end(district.second));
    }
    // sort the list -- use the list version, not the general algorithm,
    // because the list version is faster
    allNames.sort();
    // Now it's sorted, all duplicate names will be next to each other.
    // Use adjacent_find() to find instances of two or more identical names
    // next to each other.
    // Loop until adjacent_find() returns the end iterator.
    set<string> duplicates;
    for (auto lit = cbegin(allNames); lit != cend(allNames); ++lit) {
        lit = adjacent_find(lit, cend(allNames));
        if (lit == cend(allNames)) {
            break;
        }
        duplicates.insert(*lit);
    }
    return duplicates;
}
```

In this implementation, `allNames` is of type `list<string>`. That way, this example can show you how to use the `sort()` and `adjacent_find()` algorithms.

Another solution is to change the type of `allNames` to `set<string>`, which results in a more compact implementation, because a set doesn't allow duplicates. This new solution loops over all lists and tries to insert each name into `allNames`. When this insert fails, it means that there is already an element with that name in `allNames`, so the name is added to `duplicates`.

```
set<string> getDuplicates(const VotersMap& votersByDistrict)
{
    set<string> allNames;
    set<string> duplicates;
    for (auto& district : votersByDistrict) {
        for (auto& name : district.second) {
            if (!allNames.insert(name).second) {
```

```
        duplicates.insert(name);
    }
}
return duplicates;
}
```

Testing the auditVoterRolls Function

That's the complete implementation of the voter roll audit functionality. Here is a small test program:

```

// Initialize map using uniform initialization
VotersMap voters = {
    {"Orange", {"Amy Aardvark", "Bob Buffalo",
               "Charles Cat", "Dwayne Dog"}},
    {"Los Angeles", {"Elizabeth Elephant", "Fred Flamingo",
                    "Amy Aardvark"}},
    {"San Diego", {"George Goose", "Heidi Hen", "Fred Flamingo"}}
};

list<string> felons = {"Bob Buffalo", "Charles Cat"};
// Local lambda expression to print a district
auto printDistrict = [](const DistrictPair& district) {
    cout << district.first << ":";
    for (auto& str : district.second) {
        cout << " {" << str << "}";
    }
    cout << endl;
};
cout << "Before Audit:" << endl;
for_each(cbegin(voters), cend(voters), printDistrict);
cout << endl;
auditVoterRolls(voters, felons);
cout << "After Audit:" << endl;
for_each(cbegin(voters), cend(voters), printDistrict);
cout << endl;

```

The output of the program is:

Before Audit:
Los Angeles: {Elizabeth Elephant} {Fred Flamingo} {Amy Aardvark}
Orange: {Amy Aardvark} {Bob Buffalo} {Charles Cat} {Dwayne Dog}
San Diego: {George Goose} {Heidi Hen} {Fred Flamingo}

```
After Audit:  
Los Angeles: {Elizabeth Elephant}  
Orange: {Dwayne Dog}  
San Diego: {George Goose} {Heidi Hen}
```

SUMMARY

This chapter concludes the basic STL functionality. It provided an overview of the various algorithms and function objects available for your use. It also showed you how to use lambda expressions, which make it often easier to understand what your code is doing. I hope that you have

gained an appreciation for the usefulness of the STL containers, algorithms, and function objects. If not, think for a moment about rewriting the voter registration audit example without the STL. You would need to write your own linked-list and map classes, and your own searching, removing, iterating, and other algorithms. The program would be much longer, more error-prone, harder to debug, and more difficult to maintain.

The next chapters discuss a couple of other aspects of the C++ Standard Library. Chapter 18 discusses regular expressions. Chapter 19 covers a number of additional library utilities available for you to use, and Chapter 20 gives a taste of some more advanced features, such as allocators, iterator adapters, and writing your own algorithms.

18

String Localization and Regular Expressions

WHAT'S IN THIS CHAPTER?

- How to localize your applications to reach a worldwide audience
- How to use regular expressions to do powerful pattern matching

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

This chapter starts with a discussion of localization, which is becoming more and more important to allow you to write software that can be localized to different regions around the world.

The second part of this chapter introduces the *regular expressions library*, which makes it easy to perform pattern matching on strings. It allows you to search for sub-strings matching a given pattern, but also to validate, parse, and transform strings. Regular expressions are really powerful and it's recommended that you start using them instead of manually writing your own string processing code.

LOCALIZATION

When you're learning how to program in C or C++, it's useful to think of a character as equivalent to a byte and to treat all characters as members of the ASCII character set (American Standard Code for Information Interchange). ASCII is a 7-bit set usually stored in an 8-bit `char` type. In reality, experienced C++ programmers recognize that successful

programs are used throughout the world. Even if you don't initially write your program with international audiences in mind, you shouldn't prevent yourself from *localizing*, or making the software local aware, at a later date.

Localizing String Literals

A critical aspect of localization is that you should never put any native-language string literals in your source code, except maybe for debug strings targeted at the developer. In Microsoft Windows applications, this is accomplished by putting the strings in STRINGTABLE resources. Most other platforms offer similar capabilities. If you need to translate your application to another language, translating those resources should be all that needs to be done, without requiring any source changes. There are tools available that help you with this translation process.

To make your source code localizable, you should not compose sentences out of string literals, even if the individual literals can be localized. For example:

```
cout << "Read " << n << " bytes" << endl;
```

This statement cannot be localized to Dutch because it requires a reordering of the words. The Dutch translation is as follows:

```
cout << n << " bytes gelezen" << endl;
```

To make sure you can properly localize this statement, you could implement something as follows:

```
cout << Format(IDS_TRANSFERRED, n) << endl;
```

IDS_TRANSFERRED is the name of an entry in a string resource table. For the English version, IDS_TRANSFERRED could be defined as "Read \$1 bytes", while the Dutch version of the resource could be defined as "\$1 bytes gelezen". The Format() function loads the string resource and substitutes \$1 with the value of n.

Wide Characters

The problem with viewing a character as a byte is that not all languages, or *character sets*, can be fully represented in 8 bits, or 1 byte. C++ has a built-in type called wchar_t that holds a *wide character*. Languages with non-ASCII (U.S.) characters, such as Japanese and Arabic, can be represented in C++ with wchar_t. However, the C++ standard does not define a size for wchar_t. Some compilers use 16 bits while others use 32 bits. To write portable software, it is not safe to assume that wchar_t is of a particular size.

If there is *any* chance that your program will be used in a non-Western character set context (hint: there is!), you should use wide characters from the beginning. When working with wchar_t, string and character literals are prefixed with the letter L to indicate that a wide-character encoding should be used. For example, to initialize a wchar_t character to be the letter m, you write it like this:

```
wchar_t myWideCharacter = L'm';
```

There are wide-character versions of most of your favorite types and classes. The wide string class is wstring. The "prefix letter w" pattern applies to streams as well. Wide-character file output streams are handled with the wofstream, and input is handled with the wifstream. The

joy of pronouncing these class names (*woof-stream?* *whiff-stream?*) is reason enough to make your programs local aware! Streams are discussed in detail in Chapter 12.

In addition to `cout`, `cin`, `cerr`, and `clog` there are wide versions of the built-in console and error streams called `wcout`, `wcin`, `wcerr`, and `wclog`. Using them is no different than using the non-wide versions:

```
wcout << L"I am wide-character aware." << endl;
```

Non-Western Character Sets

Wide characters are a great step forward because they increase the amount of space available to define a single character. The next step is to figure out how that space is used. In wide character sets, just like in ASCII, characters are represented by numbers, now called *code points*. The only difference is that each number does not fit in 8 bits. The map of characters to code points is quite a bit larger because it handles many different character sets in addition to the characters that English-speaking programmers are familiar with.

The Universal Character Set (UCS), defined by the International Standard ISO 10646, and Unicode are both standardized sets of characters. They contain around one-hundred-thousand abstract characters, each identified by an unambiguous name and a code point. The same characters with the same numbers exist in both standards. Both have specific *encodings* that you can use. For example, UTF-8 is an example of a Unicode encoding where Unicode characters are encoded using one to four 8-bit bytes. UTF-16 encodes Unicode characters as one or two 16-bit values and UTF-32 encodes Unicode characters as exactly 32 bits.

Different applications can use different encodings. Unfortunately, the C++ standard does not specify a size for wide characters (`wchar_t`). On Windows it is 16 bits, while on other platforms it could be 32 bits. You need to be aware of this when using wide characters for character encoding in cross-platform code. To help solve this issue, there are two other character types: `char16_t` and `char32_t`. The following list gives an overview of all character types supported:

- `char`: Stores 8 bits. Can be used to store ASCII characters, or as a basic building block for storing UTF-8 encoded Unicode characters, where one Unicode character is encoded as one to four `char`s.
- `char16_t`: Stores at least 16 bits. Can be used as the basic building block for UTF-16 encoded Unicode characters where one Unicode character is encoded as one or two `char16_t`s.
- `char32_t`: Stores at least 32 bits. Can be used for storing UTF-32 encoded Unicode characters as one `char32_t`.
- `wchar_t`: Stores a wide character of a compiler-specific size and encoding.

The benefit of using `char16_t` and `char32_t` instead of `wchar_t` is that the size of `char16_t` is guaranteed to be at least 16 bits, and the size of `char32_t` is guaranteed to be at least 32 bits, independent of the compiler. There is no minimum size guaranteed for `wchar_t`.

The standard also defines the following two macros:

- `__STDC_UTF_32__`: If this is defined by the compiler, then the type `char32_t` represents a UTF-32 encoding. If it is not defined, the type `char32_t` has a compiler-dependent encoding.
- `__STDC_UTF_16__`: If this is defined by the compiler, then the type `char16_t` represents a UTF-16 encoding. If it is not defined, the type `char16_t` has a compiler-dependent encoding.

String literals can have a string prefix to turn them into a specific type. The complete set of supported string prefixes is as follows:

- `u8`: A `char` string literal with UTF-8 encoding.
- `u`: A `char16_t` string literal, which can be UTF-16 if `__STDC_UTF_16__` is defined by the compiler.
- `U`: A `char32_t` string literal, which can be UTF-32 if `__STDC_UTF_32__` is defined by the compiler.
- `L`: A `wchar_t` string literal with a compiler-dependent encoding.

All of these string literals can be combined with the raw string literal prefix, `R`, discussed in Chapter 2. For example:

```
const char* s1 = u8R"(Raw UTF-8 encoded string literal)";
const wchar_t* s2 = LR"(Raw wide string literal)";
const char16_t* s3 = uR"(Raw char16_t string literal)";
const char32_t* s4 = UR"(Raw char32_t string literal);
```

If you are using Unicode encoding, for example, by using `u8` UTF-8 string literals, or if your compiler defines `__STDC_UTF_16__` or `__STDC_UTF_32__`, you can insert a specific Unicode code point in your non-raw string literal by using the `\uABCD` notation. For example, `\u03C0` represents the PI character, and `\u00B2` represents the ² character. The following code prints "π ²":

```
const char* formula = u8"\u03C0 \u00B2";
cout << formula << endl;
```

Besides the `std::string` class, there is also support for `wstring`, `u16string`, and `u32string`. They are all defined as follows:

- `typedef basic_string<char> string;`
- `typedef basic_string<wchar_t> wstring;`
- `typedef basic_string<char16_t> u16string;`
- `typedef basic_string<char32_t> u32string;`

Multibyte characters are characters composed of one or more bytes with a compiler-dependent encoding, similar as how Unicode can be represented with one to four bytes using UTF-8, or with one or two 16-bit values using UTF-16. There are conversion functions to convert between `char16_t/char32_t` and multibyte characters, and vice versa: `mbrtoc16`, `c16rtomb`, `mbrtoc32`, and `c32rtomb`

Unfortunately, the support for `char16_t` and `char32_t` stops there. For example, the I/O stream classes in the standard library do not include support for these character types. This means that there is nothing like a version of `cout` or `cin` that supports `char16_t` and `char32_t` making it difficult to print such strings to a console or to read them from user input. If you want to do more with `char16_t` and `char32_t` strings you need to resort to third-party libraries.

Locales and Facets

Character sets are only one of the differences in data representation between countries. Even countries that use similar character sets, such as Great Britain and the United States, still differ in how they represent data such as dates and money.

The standard C++ mechanism that groups specific data about a particular set of cultural parameters is called a *locale*. An individual component of a locale, such as date format, time format, number format, etc., is called a *facet*. An example of a locale is U.S. English. An example of a facet is the format used to display a date. There are several built-in facets common to all locales. C++ also provides a way to customize or add facets.

Using Locales

When using I/O streams, data is formatted according to a particular locale. Locales are objects that can be attached to a stream. They are defined in the `<locale>` header file. Locale names can be implementation-specific. One standard is to separate a language and an area in two-letter sections with an optional encoding. For example, the locale for the English language as spoken in the U.S. is `en_US`, while the locale for the English language as spoken in Great Britain is `en_GB`. The locale for Japanese spoken in Japan with Japanese Industrial Standard encoding is `ja_JP.jis`.

Locale names on Windows follow a different standard, which has the following general format:

```
lang[_country_region[.code_page]]
```

Everything between the square brackets is optional. The following table lists some examples:

	LINUX GCC	WINDOWS
U.S. English	<code>en_US</code>	<code>English_United States</code>
Great Britain English	<code>en_GB</code>	<code>English_Great Britain</code>

Most operating systems have a mechanism to determine the locale as defined by the user. In C++, you can pass an empty string to the `locale` object constructor to create a `locale` from the user's environment. Once this object is created, you can use it to query the `locale`, possibly making programmatic decisions based on it. The following code demonstrates how to use the user's locale by calling the `imbue()` method on a stream. The result is that everything that is sent to `wcout` is formatted according to the formatting rules for your environment:

```
wcout.imbue(locale(""));
wcout << 32767 << endl;
```

This means that if your system locale is English United States and you output the number 32767, the number is displayed as 32,767; but, if your system locale is Dutch Belgium, the same number is displayed as 32.767.

The default locale is the *classic* locale, and not the user's locale. The classic locale uses ANSI C conventions, and has the name C. The classic C locale is similar to U.S. English, but there are slight differences. For example, numbers are handled without any punctuation:

```
wcout.imbue(locale("C"));
wcout << 32767 << endl;
```

The output of this code is as follows:

```
32767
```

The following code manually sets the U.S. English locale, so the number 32767 is formatted with U.S. English punctuation, independent of your system locale:

```
wcout.imbue(locale("en_US")); // Use "English_United States" on Windows
wcout << 32767 << endl;
```

The output of this code is as follows:

```
32,767
```

A `locale` object allows you to query information about the locale. For example, the following program creates a `locale` matching the user's environment. The `name()` method is used to get a C++ `string` that describes the locale. Then, the `find()` method is used on the `string` object to find a given sub-string, which returns `string::npos` when the given sub-string is not found. The code checks for the Windows name and the Linux GCC name. One of two messages is output, depending on whether the locale appears to be U.S. English or not:

```
locale loc("");
if (loc.name().find("en_US") == string::npos &&
    loc.name().find("United States") == string::npos) {
    wcout << L"Welcome non-U.S. English speaker!" << endl;
} else {
    wcout << L"Welcome U.S. English speaker!" << endl;
}
```

Using Facets

You can use the `std::use_facet()` function to obtain a particular facet in a particular locale. The argument to `use_facet()` is a `locale`. For example, the following expression retrieves the standard monetary punctuation facet of the British English locale using the Linux GCC locale name:

```
use_facet<moneypunct<wchar_t>>(locale("en_GB"));
```

Note that the innermost template type determines the character type to use. This is usually `wchar_t` or `char`. The use of nested template classes is unfortunate, but once you get past the syntax, the result is an object that contains all the information you want to know about British money punctuation.

The data available in the standard facets are defined in the `<locale>` header and its associated files.

The following program brings together locales and facets by printing out the currency symbol in both U.S. English and British English. Note that, depending on your environment, the British

currency symbol may appear as a question mark, a box, or not at all. If your environment is equipped to handle it, you may actually get the British pound symbol:

```
locale locUSEng("en_US");           // For Linux
//locale locUSEng("English_United States"); // For Windows
locale locBritEng("en_GB");         // For Linux
//locale locBritEng("English_Great Britain"); // For Windows
wstring dollars = use_facet<moneypunct<wchar_t>>(locUSEng).curr_symbol();
wstring pounds = use_facet<moneypunct<wchar_t>>(locBritEng).curr_symbol();
wcout << L"In the US, the currency symbol is " << dollars << endl;
wcout << L"In Great Britain, the currency symbol is " << pounds << endl;
```

REGULAR EXPRESSIONS

Regular expressions, defined in the `<regex>` header, are a powerful feature of the Standard Library. They are a special mini-language for string processing. They might seem complicated at first, but once you get to know them, they make working with strings easier. Regular expressions can be used for several string-related operations:

- **Validation:** Check if an input string is well-formed.
For example: Is the input string a well-formed phone number?
- **Decision:** Check what kind of string an input represents.
For example: Is the input string the name of a JPEG or a PNG file?
- **Parsing:** Extract information from an input string.
For example: From a full filename, extract the filename part without the full path and without its extension.
- **Transformation:** Search sub-strings and replace them with a new formatted sub-string.
For example: Search all occurrences of “C++14” and replace them with “C++”.
- **Iteration:** Search all occurrences of a sub-string.
For example: Extract all phone numbers from an input string.
- **Tokenization:** Split a string into sub-strings based on a set of delimiters.
For example: Split a string on whitespace, commas, periods, and so on to extract its individual words.

Of course, you could write your own code to perform any of the preceding operations on your strings, but using the regular expressions feature is highly recommended, because writing correct and safe code to process strings can be tricky.

Before we can go into more detail on the regular expressions, there is some important terminology to know. The following terms are used throughout the discussion:

- **Pattern:** The actual regular expression is a pattern represented by a string.
- **Match:** Determines whether there is a match between a given regular expression and all of the characters in a given sequence [first,last].

- **Search:** Determines whether there is some sub-string within a given sequence [first,last) that matches a given regular expression.
- **Replace:** Identifies sub-strings in a given sequence, and replaces them with a corresponding new sub-string computed from another pattern, called a *substitution pattern*.

If you look around on the internet you will find several different grammars for regular expressions. For this reason, C++ includes support for several of these grammars: *ECMAScript*, *basic*, *extended*, *awk*, *grep*, and *egrep*. If you already know any of these regular expression grammars, you can use it straight away in C++ by telling the regular expression library to use that specific syntax (`syntax_option_type`). The default grammar in C++ is *ECMAScript* whose syntax is explained in detail in the following section. It is also the most powerful grammar, so it's recommended to use *ECMAScript* instead of one of the other more limited grammars. Explaining the other regular expression grammars falls outside the scope of this book.

NOTE *If this is the first time you hear anything about regular expressions, just leave the default ECMAScript syntax.*

ECMAScript Syntax

A regular expression pattern is a sequence of characters representing what you want to match. Any character in the regular expression matches itself except for the following special characters:

`^ $ \ . * + ? () [] { } |`

These special characters are explained throughout the following discussion. If you need to match one of these special characters, you need to escape it using the `\` character. For example:

`\[or \. or * or \\`

Anchors

The special characters `^` and `$` are called *anchors*. The `^` character matches the position immediately following a line terminator character, and `$` matches the position of a line terminator character. `^` and `$` by default also match the beginning or ending of a string, respectively, but this behavior can be disabled. For example, `^test$` matches only the string `test`, and not strings that contain `test` in the line with anything else like `1test`, `test2`, `test abc`, and so on.

Wildcards

The *wildcard* character `.` can be used to match any character except a newline character. For example, the regular expression `a.c` will match `abc`, and `a5c`, but will not match `ab5c`, `ac`, and so on.

Alternation

The `|` character can be used to specify the “or” relationship. For example, `a|b` matches `a` or `b`.

Grouping

Parentheses () are used to mark *sub-expressions*, also called *capture groups*. Capture groups can be used for several purposes:

- Capture groups can be used to identify individual sub-sequences of the original string; each marked sub-expression (capture group) is returned in the result. For example, take the following regular expression: (.) (ab|cd) (.). It has three marked sub-expressions. Running a `regex_search()` with this regular expression on 1cd4 results in a match with four entries. The first entry is the entire match 1cd4 followed by three entries for the three marked sub-expressions. These three entries are 1, cd, and 4. The details on how to use the `regex_search()` algorithm are shown in a later section.
- Capture groups can be used during matching for a purpose called *back references* (explained later).
- Capture groups can be used to identify components during *replace operations* (explained later).

Repetition

Parts of a regular expression can be repeated by using one of four *repeats*:

- * matches the preceding part *zero or more* times. For example: `a*b` matches b, ab, aab, aaaab, and so on.
- + matches the preceding part *one or more* times. For example: `a+b` matches ab, aab, aaaab, and so on, but not b.
- ? matches the preceding part *zero or one* time. For example: `a?b` matches b and ab, but nothing else.
- { ... } represents a *bounded repeat*. `a{n}` matches a repeated *exactly n* times; `a{n,}` matches a repeated *n times or more*; and `a{n,m}` matches a repeated *between n and m* times inclusive. For example, `a{3,4}` matches aaa and aaaa but not a, aa, aaaa, and so on.

The repeats described in the previous list are called *greedy* because they find the longest match while still matching the remainder of the regular expression. To make them *non-greedy*, a ? can be added behind the repeat as in *?, +?, ??, and { ... }?. A non-greedy repetition repeats its pattern as few times as possible while still matching the remainder of the regular expression.

For example, the following table shows a greedy and a non-greedy regular expression and the resulting sub matches when running them on the input sequence aaabbb:

REGULAR EXPRESSION	SUB MATCHES
Greedy: (a+) (ab) * (b+)	"aaa" " " "bbb"
Non-greedy: (a+?) (ab) * (b+)	"aa" "ab" "bb"

Precedence

Just as with mathematical formulas it's important to know the precedence of regular expression elements. Precedence is as follows:

- **Elements:** like `a` are the basic building blocks of a regular expression.
- **Quantifiers:** like `+`, `*`, `?`, and `{...}` bind tightly to the element on the left; for example, `b+`.
- **Concatenation:** like `ab+c` binds after quantifiers.
- **Alternations:** like `|` binds as last.

For example, take the regular expression `ab+c|d`. This matches `abc`, `abbc`, `abbbc`, and so on, and also `d`. Parentheses can be used to change these precedence rules. For example, `ab+(c|d)` matches `abc`, `abbc`, `abbbc`, ..., `abd`, `abbd`, `abbbd`, and so on. However, by using parentheses you also mark it as a sub-expression or capture group. It is possible to change the precedence rules without creating new capture groups by using `(?:...)`. For example, `ab+(?:c|d)` matches the same as the preceding `ab+(c|d)` but does not create an additional capture group.

Character Set Matches

Instead of having to write `(a|b|c|...|z)`, which is clumsy and introduces a capture group, a special syntax for specifying sets of characters or ranges of characters is available. In addition, a “not” form of the match is also available. A *character set* is specified between square brackets, and allows you to write `[c1c2...cn]`, which matches any of the characters c_1 , c_2 , ..., or c_n . For example, `[abc]` matches any character `a`, `b`, or `c`. If the first character is `^`, it means “any but”:

- `ab[cde]` matches `abc`, `abd`, and `abe`.
- `ab[^cde]` matches `abf`, `abp`, and so on but not `abc`, `abd`, and `abe`.

If you need to match the `^`, `[` or `]` characters themselves, you need to escape them; for example: `[\^\[]` matches the characters `[`, `^` or `]`.

If you want to specify all letters, you could use a character set like

`[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]`; however, this is clumsy and doing this several times is awkward, especially if you make a typo and omit one of the letters accidentally. There are two solutions to this.

The *range specification* in square brackets allows you to write `[a-zA-Z]`, which recognizes all the letters in the range `a` to `z` and `A` to `Z`. If you need to match a hyphen, you need to escape it; for example, `[a-zA-Z\ -]+` matches any word including a hyphenated word.

Another capability is to use one of the *character classes*. These are used to denote specific types of characters and are represented as `[:name:]`. Which character classes are available depends on the locale, but the names listed in the following table are always recognized. The exact meaning of these character classes is also dependent on the locale. This table assumes the standard C locale.

CHARACTER CLASS NAME	DESCRIPTION
digit	Digits.
d	Same as digit.
xdigit	Digits (digit) and the following letters used in hexadecimal numbers 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'
alpha	Alphabetic characters. For the C locale these are all lowercase and uppercase letters.
alnum	A combination of the alpha class and the digit class.
w	Same as alnum.
lower	Lowercase letters, if applicable to the locale.
upper	Uppercase letters, if applicable to the locale.
blank	A blank character is a space character used to separate words within a line of text. For the C locale these are ' ' or '\t' (tab).
space	Whitespace characters. For the C locale, these are ' ', '\t', '\n', '\r', '\v', and '\f'.
s	Same as space.
print	Printable characters. These occupy a printing position, for example, on a display, and are the opposite of control characters (cntrl). Examples are lowercase letters, uppercase letters, digits, punctuation characters, and space characters.
cntrl	Control characters. These are the opposite of printable characters (print), and don't occupy a printing position, for example, on a display. Some examples for the C locale are '\f' (form feed), '\n' (new line), and '\r' (carriage return).
graph	Characters with a graphical representation. These are all characters that are printable (print), except the space character ' '.
punct	Punctuation characters. For the C locale, these are all graphical characters (graph) that are not alphanumeric (alnum).

Character classes are used within character sets; for example, `[:alpha:]*` in English means the same as `[a-zA-Z]*`.

Because certain concepts like matching digits are so common, there are shorthand patterns for them. For example, `[:digit:]` and `[:d:]` mean the same thing as `[0-9]`. Some classes have an

even shorter pattern using the escape notation `\d`. For example `\d` means `[:digit:]`. Therefore, to recognize a sequence of one or more numbers, you can write any of the following patterns:

- `[0-9] +`
- `[[:digit:]] +`
- `[[:d:]] +`
- `\d +`

The following table lists the available escape notations for character classes:

ESCAPE NOTATION	EQUIVALENT TO
<code>\d</code>	<code>[[:d:]]</code>
<code>\D</code>	<code>[[^:d:]]</code>
<code>\s</code>	<code>[[:s:]]</code>
<code>\S</code>	<code>[[^:s:]]</code>
<code>\w</code>	<code>[_[:w:]]</code>
<code>\W</code>	<code>[_[^:w:]]</code>

Some examples:

- `Test[5-8]` matches `Test5`, `Test6`, `Test7`, and `Test8`.
- `[:lower:]` matches `a`, `b`, and so on but not `A`, `B`, and so on.
- `[[^:lower:]]` matches any character except lowercase letters like `a`, `b`, and so on.
- `[:lower:]5-7` matches any lower case letter like `a`, `b`, and so on and the numbers `5`, `6`, and `7`.

Word Boundaries

A *word boundary* can mean the following:

- The beginning of the source string if the first character of the source string is one of the word characters `[A-Za-z0-9_]`. Matching the beginning of the source string is enabled by default, but you can disable it (`regex_constants::match_not_bow`).
- The end of the source string if the last character of the source string is one of the word characters. Matching the end of the source string is enabled by default, but you can disable it (`regex_constants::match_not_eow`).
- The first character of a word, which is one of the word characters, while the preceding character is not a word character.
- The end of a word, which is a non-word character after a word, while the preceding character is a word character.

You can use `\b` to match a word boundary, and `\B` to match anything except a word boundary.

Back References

Back references allow you to reference a captured group inside the regular expression itself: `\n` refers to the n -th captured group, with $n > 0$. For example, the regular expression `(\d+) - .*-\1` matches a string that has the following format:

- one or more digits captured in a capture group `(\d+)`
- followed by a dash -
- followed by zero or more characters `.*`
- followed by another dash -
- followed by exactly the same digits captured by the first capture group `\1`

This regular expression matches `123-abc-123`, `1234-a-1234`, and so on but does not match `123-abc-1234`, `123-abc-321`, and so on.

Lookahead

Regular expressions support *positive lookahead* (`(?=pattern)`) and *negative lookahead* (`(?!pattern)`). The characters following the lookahead must match (positive) or not match (negative) the lookahead pattern, but those characters are not yet consumed. For example, the following regular expression matches an input sequence that consists of at least one lower case letter, at least one upper case letter, at least one punctuation character, and is at least eight characters long:

```
(?=.*[:lower:]).(?=.*[:upper:]).(?=.*[:punct:]).{8,}
```

Regular Expressions and Raw String Literals

As seen in the preceding sections, regular expressions often use special characters that should be escaped in normal C++ string literals. For example, if you write `\d` in a regular expression it matches any digit. However, since `\` is a special character in C++, you need to escape it in your regular expression string literal as `\\\d`, otherwise your C++ compiler tries to interpret the `\d`. It can get more complicated if you want your regular expression to match a single back-slash character `\`. Because `\` is a special character in the regular expression syntax itself, you need to escape it as `\\\`. The `\` character is also a special character in C++ string literals, so you need to escape it in your C++ string literal, resulting in `\\\\\`.

You can use raw string literals to make complicated regular expression easier to read in your C++ source code. Raw string literals are discussed in Chapter 2. For example take the following regular expression:

```
" ( |\\n|\\r|\\\\\\ )"
```

This regular expression searches for spaces, newlines, carriage returns, and back slashes. As you can see, you need a lot of escape characters. Using raw string literals, this can be replaced with the following more readable regular expression:

```
R"(( |\\n|\\r|\\\\ ))"
```

The raw string literal starts with `R" (` and ends with `) "`. Everything in between is the regular expression. Of course you still need a double back slash at the end because the back slash needs to be escaped in the regular expression itself.

This concludes a brief description of the ECMAScript grammar. The following section starts with actually using regular expressions in your C++ code.

The regex Library

Everything for the regular expression library is in the `<regex>` header file and in the `std` namespace. The basic templated types defined by the regular expression library are:

- `basic_regex`: An object representing a specific regular expression.
- `match_results`: A sub-string that matched a regular expression, including all the captured groups. It is a collection of `sub_matches`.
- `sub_match`: An object containing a pair of iterators into the input sequence. These iterators represent the matched capture group. The pair is an iterator pointing to the first character of a matched capture group and an iterator pointing to one-past-the-last character of the matched capture group. It has a `str()` method that returns the matched capture group as a string.

The library provides three key algorithms: `regex_match()`, `regex_search()`, and `regex_replace()`. All of these algorithms have different versions that allow you to specify the source string as an STL string, a character array, or as a begin/end iterator pair. The iterators can be any of the following:

- `const char*`
- `const wchar_t*`
- `string::const_iterator`
- `wstring::const_iterator`

In fact, any iterator that behaves as a bidirectional iterator can be used. Iterators are discussed in detail in Chapter 16.

The library also defines *regular expression iterators*, which are very important if you want to find all occurrences of a pattern in a source string. There are two templated regular expression iterators defined:

- `regex_iterator`: iterates over all the occurrences of a pattern in a source string
- `regex_token_iterator`: iterates over all the capture groups of all occurrences of a pattern in a source string

To make the library easier to use, the standard defines a number of `typedefs` for the preceding templates:

```
typedef basic_regex<char>      regex;
typedef basic_regex<wchar_t>    wregex;
typedef sub_match<const char*>      csub_match;
typedef sub_match<const wchar_t*>    wcsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;
typedef match_results<const char*>      cmatch;
typedef match_results<const wchar_t*>    wcmatch;
```

```

typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<string::const_iterator> sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;

```

The following sections explain the `regex_match()`, `regex_search()`, and `regex_replace()` algorithms, and the `regex_iterator` and `regex_token_iterator`.

regex_match()

The `regex_match()` algorithm can be used to compare a given source string with a regular expression pattern and returns `true` if the pattern matches the entire source string, `false` otherwise. It is very easy to use. There are six versions of the `regex_match()` algorithm accepting different kinds of arguments. They all have the following form:

```

template<...>
bool regex_match(InputSequence[, MatchResults], RegEx[, Flags]);

```

All variations return `true` when the entire input sequence matches the pattern, `false` otherwise. The `InputSequence` can be represented as:

- A start and end iterator into a source string
- A `std::string`
- A C-style string

The optional `MatchResults` parameter is a reference to a `match_results` and receives the match. If `regex_match()` returns `false`, you are only allowed to call `match_results::empty()` or `match_results::size()`; anything else is undefined. If `regex_match()` returns `true`, a match is found and you can inspect the `match_results` object for what exactly got matched. How to do this is explained with examples in the following sections.

The `RegEx` parameter is the regular expression that needs to be matched. The optional `Flags` parameter specifies options for the matching algorithm. In most cases you can keep the default. Consult a Standard Library Reference — for example <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/> — for more details.

regex_match() Example

Suppose you want to write a program that asks the user to enter a date in the following format year/month/day where year is four digits, month is a number between 1 and 12, and day is a number between 1 and 31. You can use a regular expression together with the `regex_match()` algorithm to validate the user input as follows. The details of the regular expression are explained after the code:

```

regex r("\d{4}/(?:0?[1-9]|1[0-2])/(?:0?[1-9]|1[1-2]\ [0-9]|3[0-1])");
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";

```

```
string str;
if (!getline(cin, str) || str == "q")
    break;
if (regex_match(str, r))
    cout << "  Valid date." << endl;
else
    cout << "  Invalid date!" << endl;
}
```

The first line creates the regular expression. The expression consists of three parts separated by a forward slash / character, one part for year, one for month, and one for day. The following list explains these parts:

- `\d{4}`: This matches any combination of four digits; for example, 1234, 2010, and so on.
- `(?:0?[1-9] | 1[0-2])`: This sub part of the regular expression is wrapped inside parentheses to make sure the precedence is correct. We don't need any capture group so `(?:...)` is used. The inner expression consists of an alternation of two parts separated by the `|` character.
 - `0?[1-9]`: This matches any number from 1 to 9 with an optional 0 in front of it. For example, it matches 1, 2, 9, 03, 04, and so on. It does not match 0, 10, 11, and so on.
 - `1[0-2]`: This matches 10, 11, or 12, and nothing else.
- `(?:0?[1-9] | [1-2][0-9] | 3[0-1])`: This sub part is also wrapped inside a non-capture group and consists of an alternation of three parts:
 - `0?[1-9]`: This matches any number from 1 to 9 with an optional 0 in front of it. For example, it matches 1, 2, 9, 03, 04, and so on. It does not match 0, 10, 11, and so on.
 - `[1-2][0-9]`: This matches any number between 10 and 29 inclusive and nothing else.
 - `3[0-1]`: This matches 30 or 31 and nothing else.

The example then enters an infinite loop to ask the user to enter a date. Each date entered is given to the `regex_match()` algorithm. When `regex_match()` returns `true` the user has entered a date that matches the date regular expression pattern.

This example can be expanded a bit by asking the `regex_match()` algorithm to return captured sub-expressions in a `results` object. The following code extracts the year, month, and day digits into three separate integer variables.

To understand this code, you have to understand what a capture group does. By specifying a `match_results` object like `smatch` in the call to `regex_match()`, the elements of the `match_results` object are filled in when the regular expression matches the string. To be able to extract these sub-strings, you must create capture groups, so parentheses are used to define new capture groups.

The first element, `[0]`, in a `match_results` object contains the string that matched the entire pattern. When using `regex_match()` and a match is found, this is the entire source sequence. When using `regex_search()`, discussed in the next section, this is a sub-string in the source sequence that matches the regular expression. Element `[1]` is the sub-string matched by the first capture group, `[2]` by the second capture group, and so on. To get a string representation of a capture group, you

can write `m[i]` as in the following code or write `m[i].str()`, where `i` is the index of the capture group.

The regular expression in the revised example has a few small changes. The first part matching the year is wrapped in a capture group, while the month and day parts are now also capture groups instead of non-capture groups. The call to `regex_match()` includes a `smatch` parameter, which receives the matched capture groups. Here is the adapted example:

```
regex r("(\\d{4})/(0?[1-9]|1[0-2])/([0?[1-9]|1-2][0-9]|3[0-1])");
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    smatch m;
    if (regex_match(str, m, r)) {
        int year = stoi(m[1]);
        int month = stoi(m[2]);
        int day = stoi(m[3]);
        cout << "  Valid date: Year=" << year
            << ", month=" << month
            << ", day=" << day << endl;
    } else {
        cout << "  Invalid date!" << endl;
    }
}
```

In this example, there are four elements in the `smatch` results objects: the full match and three captured groups:

- [0]: the string matching the full regular expression, which is the full date in this example
- [1]: the year
- [2]: the month
- [3]: the day

When you execute this example you can get the following output:

```
Enter a date (year/month/day) (q=quit): 2011/12/01
    Valid date: Year=2011, month=12, day=1
Enter a date (year/month/day) (q=quit): 11/12/01
    Invalid date!
```

NOTE *These date-matching examples only check if the date consists of a year (four digits), a month (1-12), and a day (1-31). They do not perform any validation for leap years and so on. If you need that, you have to write code to validate the year, month and day values that are extracted by `regex_match()`. This validation is not a job for regular expressions, so this is not shown.*

regex_search()

The `regex_match()` algorithm discussed in the previous section returns `true` if the entire source string matches the regular expression, `false` otherwise. It cannot be used to find a matching sub-string in the source string. The `regex_search()` algorithm allows you to search for a sub-string that matches a certain pattern in a source string. There are six versions of the `regex_search()` algorithm. They all have the following form:

```
template<...>
bool regex_search(InputSequence[, MatchResults], RegEx[, Flags]);
```

All variations return `true` when a match is found in the input sequence, `false` otherwise. The `InputSequence` can be represented as:

- A start and end iterator into a source string
- A `std::string`
- A C-style string

The optional `MatchResults` parameter is a reference to a `match_results` and receives the match. If `regex_search()` returns `false`, you are only allowed to call `match_results::empty()` or `match_results::size()`; anything else is undefined. If `regex_search()` returns `true`, a match is found and you can inspect the `match_results` object for what exactly got matched.

The `RegEx` parameter is the regular expression that needs to be matched. The optional `Flags` parameter specifies options for the matching algorithm. In most cases you can keep the default. Consult a Standard Library Reference for more details.

Two versions of the `regex_search()` algorithm accept a begin and end iterator as the input sequence that you want to process. You might be tempted to use this version of `regex_search()` in a loop to find all occurrences of a pattern in a source string by manipulating these begin and end iterators for each `regex_search()` call. Never do this! It can cause problems when your regular expression uses anchors (^ or \$), word boundaries, and so on. It can also cause an infinite loop due to empty matches. Use the `regex_iterator` or `regex_token_iterator` as explained later in this chapter to extract all occurrences of a pattern from a source string.

WARNING *Never use `regex_search()` in a loop to find all occurrences of a pattern in a source string. Instead, use a `regex_iterator` or `regex_token_iterator`.*

regex_search() Example

The `regex_search()` algorithm can be used to extract matching sub-strings from an input sequence. The following example extracts code comments from input lines. The regular expression searches for a sub-string that starts with `//` followed by some optional whitespace `\s*` followed by one or more characters captured in a capture group `(.+)`. This capture group captures only the comment sub-string. The `smatch` object `m` receives the search results. You can check the `m[1].first` and `m[1].second` iterators to see where exactly the sub-string matching the first capture group was found in the source string.

```

regex r("//\\s*(.+)\\$");
while (true) {
    cout << "Enter a string with optional code comments (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    smatch m;
    if (regex_search(str, m, r))
        cout << "  Found comment '" << m[1] << "'";
    else
        cout << "  No comment found!" << endl;
}

```

The output of this program can look as follows:

```

Enter a string (q=quit): std::string str; // Our source string
    Found comment 'Our source string'
Enter a string (q=quit): int a; // A comment with // in the middle
    Found comment 'A comment with // in the middle'
Enter a string (q=quit): float f; // A comment with a      (tab) character
    Found comment 'A comment with a      (tab) character'

```

The `match_results` object also has a `prefix()` and `suffix()` method, which returns the string preceding or following the match respectively.

regex_iterator

As explained in the previous section, you should never use `regex_search()` in a loop to extract all occurrences of a pattern from a source sequence. Instead, you should use a `regex_iterator` or `regex_token_iterator`. They work similarly like iterators for STL containers, which are discussed in Chapter 16.

Internally, both a `regex_iterator` and a `regex_token_iterator` contain a pointer to the regular expression. Because of this, you should not create them with a temporary `regex` object.

WARNING *Never try to create a `regex_iterator` or `regex_token_iterator` with a temporary `regex` object.*

regex_iterator Example

The following example asks the user to enter a source string, extracts every word from the string, and prints all words between quotes. The regular expression in this case is `[\w]+`, which searches for one or more word-letters. This example uses `std::string` as source, so it uses `sregex_iterator` for the iterators. A standard iterator loop is used, but in this case, the end iterator is done slightly differently from the end iterators of ordinary STL containers. Normally, you specify an end iterator for a particular container, but for `regex_iterator`, there is only one “end” value. You can get this end iterator by declaring a `regex_iterator` type using the default constructor; it will implicitly be initialized to the end value.

The `for` loop creates a start iterator called `iter`, which accepts a begin and end iterator into the source string together with the regular expression. The loop body is called for every match found, which is

every word in this example. The `sregex_iterator` iterates over all the matches. By dereferencing a `sregex_iterator`, you get a `smatch` object. Accessing the first element of this `smatch` object, `[0]`, gives you the matched sub-string:

```
regex reg("[\\w]+");
while (true) {
    cout << "Enter a string to split (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    const sregex_iterator end;
    for (sregex_iterator iter(cbegin(str), cend(str), reg);
        iter != end; ++iter) {
        cout << "\"" << (*iter)[0] << "\" " << endl;
    }
}
```

The output of this program can look as follows:

```
Enter a string to split (q=quit): This, is      a test.
"This"
"is"
"a"
"test"
```

As this example demonstrates, even simple regular expressions can do some powerful string manipulation.

regex_token_iterator

The previous section describes `regex_iterator`, which iterates through every matched pattern. In each iteration of the loop you get a `match_results` object, which you can use to extract sub-expressions for that match captured by capture groups.

A `regex_token_iterator` can be used to automatically iterate over all or selected capture groups across all matched patterns. There are four constructors with the following format:

```
regex_token_iterator(BidirectionalIterator a,
                    BidirectionalIterator b,
                    const regex_type& re
                    [, SubMatches
                    [, Flags]]);
```

All of them require a begin and end iterator as input sequence, and a regular expression. The optional `SubMatches` parameter is used to specify which capture groups should be iterated over. `SubMatches` can be specified in four ways:

- A single integer representing the index of the capture group that you want to iterate over.
- A vector with integers representing the indices of the capture groups that you want to iterate over.
- An `initializer_list` with capture group indices.
- A C-style array with capture group indices.

When you omit `SubMatches` or when you specify a 0 for `SubMatches`, you get an iterator that iterates over all capture groups with index 0, which are the sub-strings matching the full regular expression. The optional `Flags` parameter specifies options for the matching algorithm. In most cases you can keep the default. Consult a Standard Library Reference for more details.

regex_token_iterator Examples

The previous `regex_iterator` example can be rewritten using a `regex_token_iterator` as follows. Note that `*iter` is used in the loop body instead of `(*iter)[0]` as in the `regex_iterator` example because the token iterator with 0 as the default `submatch` index automatically iterates over all capture groups with index 0. The output of this code is exactly the same as the output generated by the `regex_iterator` example:

```
regex reg("[\\w]+");
while (true) {
    cout << "Enter a string to split (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    const sregex_token_iterator end;
    for (sregex_token_iterator iter(cbegin(str), cend(str), reg);
         iter != end; ++iter) {
        cout << "\"" << *iter << "\""
        << endl;
    }
}
```

The following example asks the user to enter a date and then uses a `regex_token_iterator` to iterate over the second and third capture groups (month and day), which are specified as a vector of integers. The regular expression used for dates is explained earlier in this chapter. The only difference is that `^` and `$` anchors are added, which are not necessary earlier because that example uses `regex_match()`.

```
regex reg("^(\\d{4})/(0?[1-9]|1[0-2])/((0?[1-9]|1[1-2]) [0-9]|3[0-1])$");
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    vector<int> indices{ 2, 3 };
    const sregex_token_iterator end;
    for (sregex_token_iterator iter(cbegin(str), cend(str), reg, indices);
         iter != end; ++iter) {
        cout << "\"" << *iter << "\""
        << endl;
    }
}
```

This code prints only the month and day of valid dates. Output generated by this example can look as follows:

```
Enter a date (year/month/day) (q=quit): 2011/1/13
"1"
"13"
Enter a date (year/month/day) (q=quit): 2011/1/32
```

```
Enter a date (year/month/day) (q=quit): 2011/12/5
"12"
"5"
```

The `regex_token_iterator` can also be used to perform a so-called *field splitting* or *tokenization*. It is a much safer and more flexible alternative than using the old `strtok()` function from C. Tokenization is triggered in the `regex_token_iterator` constructor by specifying `-1` as the capture group index to iterate over. When in tokenization mode, the iterator iterates over all sub-strings of the input sequence that **do not match** the regular expression. The following code demonstrates this by tokenizing a string on the delimiters `,` and `;` with any number of whitespace characters before or after the delimiters:

```
regex reg(R"(\s*[,;]\s*)");
while (true) {
    cout << "Enter a string to split on ',' and ';' (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q")
        break;
    const sregex_token_iterator end;
    for (sregex_token_iterator iter(cbegin(str), cend(str), reg, -1);
         iter != end; ++iter) {
        cout << "\"" << *iter << "\""
        << endl;
    }
}
```

The regular expression in this example is specified as a raw string literal and searches for patterns that match the following:

- Zero or more whitespace characters,
- followed by a `,` or `;` character,
- followed by zero or more whitespace characters.

The output can be as follows:

```
Enter a string to split on ',' and ';' (q=quit): This is,    a; test string.
"This is"
"a"
"test string."
```

As you can see from this output, the string is split on `,` and `;`. All whitespace characters around the `,` or `;` are removed, because the tokenization iterator iterates over all sub-strings that **do not match** the regular expression, and because the regular expression matches `,` and `;` with whitespace around them.

regex_replace()

The `regex_replace()` algorithm requires a regular expression, and a formatting string that is used to replace matching sub-strings. This formatting string can reference part of the matched sub-strings by using the following escape sequences:

ESCAPE SEQUENCE	REPLACED WITH
<code>\$n</code>	the string matching the <i>n</i> -th capture group; for example, <code>\$1</code> for the first capture group, <code>\$2</code> for the second, and so on
<code>\$&</code>	the string matching the whole regular expression, which is the same as <code>\$0</code>
<code>\$^</code>	the part of the input sequences that appears to the left of the sub-string matching the regular expression
<code>\$'</code>	the part of the input sequence that appears to the right of the sub-string matching the regular expression
<code>\$\$</code>	a dollar sign

There are six versions of the `regex_replace()` algorithm. The difference between them is in the type of arguments. Four of them have the following format:

```
string regex_replace(InputSequence, RegEx, FormatString[, Flags]);
```

These four versions return the resulting string after performing the replacement. The `InputSequence` can be a `std::string` or a C-style string. The `RegEx` parameter is the regular expression that needs to be matched. The `FormatString` can be a `std::string` or a C-style string. The optional `Flags` parameter specifies options for the replace algorithm.

Two versions of the `regex_replace()` algorithm have the following format:

```
OutputIterator regex_replace(OutputIterator,
                           BidirectionalIterator first,
                           BidirectionalIterator last,
                           RegEx, FormatString[, Flags]);
```

These two versions write the resulting string to the given output iterator and return this output iterator. The input sequence is given as a begin and end iterator. The other parameters are identical to the other four versions of `regex_replace()`.

regex_replace() Examples

As a first example, take the source HTML string `<body><h1>Header</h1><p>Some text</p></body>` and the regular expression `<h1>(.*)</h1><p>(.*)</p>`. The following table shows the different escape sequences and what they will be replaced with:

ESCAPE SEQUENCE	REPLACED WITH
<code>\$1</code>	Header
<code>\$2</code>	Some text
<code>\$&</code>	<code><h1>Header</h1><p>Some text</p></code>
<code>\$^</code>	<code><body></code>
<code>\$'</code>	<code></body></code>

The following code demonstrates the use of `regex_replace()`:

```
const string str("<body><h1>Header</h1><p>Some text</p></body>");  
regex r("<h1>(.*)</h1><p>(.*)</p>");  
const string format("H1=$1 and P=$2");  
string result = regex_replace(str, r, format);  
cout << "Original string: '" << str << "'" << endl;  
cout << "New string      : '" << result << "'" << endl;
```

The output of this program is as follows:

```
Original string: '<body><h1>Header</h1><p>Some text</p></body>'  
New string      : '<body>H1=Header and P=Some text</body>'
```

The `regex_replace()` algorithm accepts a number of flags that can be used to manipulate how it is working. The most important flags are given in the following table:

FLAG	DESCRIPTION
<code>format_default</code>	The default is to replace all occurrences of the pattern, and to also copy everything that does not match the pattern to the result string.
<code>format_no_copy</code>	Replace all occurrences of the pattern, but do not copy anything that does not match the pattern to the result string.
<code>format_first_only</code>	Replace only the first occurrence of the pattern.

The following example modifies the previous code to use the `format_no_copy` flag:

```
const string str("<body><h1>Header</h1><p>Some text</p></body>");  
regex r("<h1>(.*)</h1><p>(.*)</p>");  
const string format("H1=$1 and P=$2");  
string result = regex_replace(str, r, format,  
    regex_constants::format_no_copy);  
cout << "Original string: '" << str << "'" << endl;  
cout << "New string      : '" << result << "'" << endl;
```

The output is as follows. Compare this with the output of the previous version.

```
Original string: '<body><h1>Header</h1><p>Some text</p></body>'  
New string      : 'H1=Header and P=Some text'
```

Another example is to get an input string and replace each word boundary with a newline so that the target string contains only one word per line. The following example demonstrates this without using any loops to process a given string. The code first creates a regular expression that matches individual words. When a match is found it is replaced with `$1\n` where `$1` is replaced with the matched word. Note also the use of the `format_no_copy` flag to prevent copying whitespace and other non-word characters from the source string to the result string:

```
regex reg("([\w]+)");  
const string format("$1\n");  
while (true) {  
    cout << "Enter a string to split over multiple lines (q=quit): ";  
    string str;
```

```
if (!getline(cin, str) || str == "q")
    break;
cout << regex_replace(str, reg, format,
    regex_constants::format_no_copy) << endl;
}
```

The output of this program can be as follows:

```
Enter a string to split over multiple lines (q=quit): This is a test.
This
is
a
test
```

SUMMARY

This chapter gave you an appreciation for coding with localization in mind. As anyone who has been through a localization effort will tell you, adding support for a new language or locale is infinitely easier if you have planned ahead; for example, by using Unicode characters and being mindful of locales.

The second part of this chapter explained the regular expressions library. Once you know the syntax of regular expressions, it becomes much easier to work with strings. Regular expressions allow you to validate strings, search for sub-strings inside an input sequence, perform find-and-replace operations, and so on. It is highly recommended you get to know them and start using them instead of writing your own string manipulation routines. They will make your life easier.

19

Additional Library Utilities

WHAT'S IN THIS CHAPTER?

- How you can use `std::function` for function pointers
- How to work with compile-time rational numbers
- How to work with time
- How to generate random numbers
- What tuples are and how to use them

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

This chapter discusses some additional library functionality that is available in the C++ standard library and that does not immediately fit into other chapters.

STD::FUNCTION

`std::function`, defined in the `<functional>` header file, can be used to create a type that can point to a function, a function object, or a lambda expression; basically anything that is callable. It is called a *polymorphic function wrapper*. It can be used as a function pointer or as a parameter for a function to implement callbacks. The template parameters for the `std::function` template look a bit different than most template parameters. Its syntax is as follows:

```
std::function<R(ArgTypes...)>
```

`R` is the return value type of the function and `ArgTypes` is a comma-separated list of argument types for the function.

The following example demonstrates how to use `std::function` to implement a function pointer. It creates a function pointer `f1` to point to the function `func()`. Once `f1` is defined, you can call `func()` by using the name `func` or `f1`:

```
void func(int num, const string& str)
{
    cout << "func(" << num << ", " << str << ")" << endl;
}
int main()
{
    function<void(int, const string&)> f1 = func;
    f1(1, "test");
    return 0;
}
```

Of course, in the preceding example it is possible to use the `auto` keyword, which removes the need for you to specify the exact type of `f1`. The following works exactly the same and is much shorter, but the compiler-deduced type of `f1` is a function pointer, i.e. `void (*f1)(int, const string&)`, instead of a `std::function`.

```
auto f1 = func;
```

Since `std::function` types behave as function pointers, they can be passed to STL algorithms as shown in the following example using the `count_if()` algorithm. STL algorithms are discussed in Chapter 17.

```
bool isEven(int num)
{
    return num % 2 == 0;
}
int main()
{
    vector<int> vec(10);
    iota(begin(vec), end(vec), 0);
    function<bool(int)> f2 = isEven;
    int cnt = count_if(cbegin(vec), cend(vec), f2);
    cout << cnt << " even numbers" << endl;
    return 0;
}
```

After the preceding examples, you might think that `std::function` is not really useful; but, where `std::function` really shines, is accepting a function pointer as parameter to your own function. The following example defines a function called `process()`, which accepts a reference to a `vector` and a `std::function`. The `process()` function iterates over all the elements in the given `vector` and calls the given function `f` for each element. You can think of the parameter `f` as a callback.

The `print()` function prints a given element to the console. The `main()` function first creates a `vector` of integers and populates it. It then calls the `process()` function with a function pointer to `print()`. The result is that each element in the `vector` is printed.

The last part of the `main()` function demonstrates that you can also pass a lambda expression for the `std::function` parameter of the `process()` function, and that's the power of `std::function`. You cannot get this same functionality by using a function pointer `typedef`.

```
void process(const vector<int>& vec, function<void(int)> f)
{
    for (auto& i : vec) {
        f(i);
    }
}
void print(int num)
{
    cout << num << " ";
}
int main()
{
    vector<int> vec(10);
    iota(begin(vec), end(vec), 0);
    process(vec, print);
    cout << endl;
    int sum = 0;
    process(vec, [&sum](int num){sum += num;});
    cout << "sum = " << sum << endl;
    return 0;
}
```

The output of this example is as follows:

```
0 1 2 3 4 5 6 7 8 9
sum = 45
```

RATIOS

The Ratio library allows you to exactly represent any finite rational number that you can use at compile time. Ratios are used in the `std::chrono::duration` class discussed in the next section. Everything is defined in the `<ratio>` header file and is in the `std` namespace. The numerator and denominator of a rational number are represented as compile-time constants of type `std::intmax_t`, which is a signed integer type with the maximum width supported by a compiler. Because of the compile-time nature of these rational numbers, using them might look a bit complicated and different than usual. You cannot define a `ratio` object the same way as you define normal objects, and you cannot call methods on it. You need to use `typedefs`. For example, the following line defines a rational compile-time constant representing 1/60:

```
typedef ratio<1, 60> r1;
```

The numerator and denominator of the `r1` rational number are compile-time constants and can be accessed as follows:

```
intmax_t num = r1::num;
intmax_t den = r1::den;
```

Remember that a `ratio` is a **compile-time constant**, which means that the numerator and denominator need to be known at compile time. The following will generate a compiler error:

```
intmax_t n = 1;
intmax_t d = 60;
typedef ratio<n, d> r1;      // Error
```

Making `n` and `d` constants removes the compilation error:

```
const intmax_t n = 1;
const intmax_t d = 60;
typedef ratio<n, d> r1;      // Ok
```

Rational numbers are always normalized. For a rational number `ratio<n, d>`, the greatest common divisor, `gcd`, is calculated and the numerator, `num`, and denominator, `den`, are defined as follows:

- `num = sign(n)*sign(d)*abs(n)/gcd`
- `den = abs(d)/gcd`

The library supports adding, subtracting, multiplying, and dividing rational numbers. Because all these operations are also happening at compile time, you cannot use the standard arithmetic operators.

Instead, you need to use specific templates in combination with `typedefs`. The following arithmetic `ratio` templates are available: `ratio_add`, `ratio_subtract`, `ratio_multiply`, and `ratio_divide`.

These templates calculate the result as a new `ratio` type. This type can be accessed with the embedded `typedef` called `type`. For example, the following code first defines two `ratios`, one representing 1/60 and the other representing 1/30. The `ratio_add` template adds those two rational numbers together to produce the `result` rational number, which, after normalization, is 1/20:

```
typedef ratio<1, 60> r1;
typedef ratio<1, 30> r2;
typedef ratio_add<r1, r2>::type result;
```

The standard also defines a number of `ratio` comparison templates: `ratio_equal`, `ratio_not_equal`, `ratio_less`, `ratio_less_equal`, `ratio_greater`, and `ratio_greater_equal`. Just like the arithmetic `ratio` templates, the `ratio` comparison templates are all evaluated at compile time. These comparison templates create a new type, `std::integral_constant`, representing the result. An `integral_constant` is a `struct` template that stores a type and a compile-time constant value. For example, `integral_constant<bool, true>` stores a Boolean with value `true`, while `integral_constant<int, 15>` stores an integer with value `15`. The result of the `ratio` comparison templates is either `integral_constant<bool, true>` or `integral_constant<bool, false>`. The value associated with an `integral_constant` can be accessed using the `value` data member. The following example demonstrates the use of `ratio_less`. Chapter 12 discusses the use of `boolalpha` to output `true` or `false` for Boolean values:

```
typedef ratio<1, 60> r1;
typedef ratio<1, 30> r2;
typedef ratio_less<r2, r1> res;
cout << boolalpha << res::value << endl;
```

The following example combines everything. Note that since ratios are compile-time constants, you cannot do something like `cout << r1`, you need to get the numerator and denominator and print them separately:

```

// Define a compile-time rational number
typedef ratio<1, 60> r1;
cout << "1) " << r1::num << "/" << r1::den << endl;
// Get numerator and denominator
intmax_t num = r1::num;
intmax_t den = r1::den;
cout << "2) " << num << "/" << den << endl;
// Add two rational numbers
typedef ratio<1, 30> r2;
cout << "3) " << r2::num << "/" << r2::den << endl;
typedef ratio_add<r1, r2>::type result;
cout << "4) " << result::num << "/" << result::den << endl;
// Compare two rational numbers
typedef ratio_less<r2, r1> res;
cout << "5) " << boolalpha << res::value << endl;

```

The output should be as follows:

- 1) $1/60$
 - 2) $1/60$
 - 3) $1/30$
 - 4) $1/20$
 - 5) false

The library provides a number of SI (*Système International*) typedefs for your convenience. They are as follows:

The SI units with an asterisk at the end are defined only if your compiler can represent the constant numerator and denominator values for those `typedefs` as an `intmax_t`. An example on how to use these predefined SI units is given during the discussion of durations in the next section.

THE CHRONO LIBRARY

The Chrono library is a collection of classes to work with times. The library consists of the following components:

- Duration
- Clock
- Time point

Everything is defined in the `std::chrono` namespace and requires you to include the `<chrono>` header file. The following sections explain each component.

Duration

A *duration*, which represents an interval between two points in time, is specified by the templated duration class. The duration class stores a number of *ticks* and a *tick period*. The tick period is the time in seconds between two ticks and is represented as a compile-time `ratio` constant, which means it could be a fraction of a second. Ratios are discussed in the previous section. The `duration` template accepts two template parameters and is defined as follows:

```
template <class Rep, class Period = ratio<1>> class duration {...}
```

The first template parameter, `Rep`, is the type of variable storing the number of ticks and should be an arithmetic type, for example `long`, `double`, and so on. The second template parameter, `Period`, is the rational constant representing the period of a tick. If you don't specify the tick period, the default value `ratio<1>` is used, which represents a tick period of 1 second.

Three constructors are provided: the default constructor; one that accepts a single value, the number of ticks; and one that accepts another duration. The latter can be used to convert from one duration to another duration, for example from minutes to seconds. An example is given later in this section.

Durations support arithmetic operations such as `+`, `-`, `*`, `/`, `%`, `++`, `--`, `+=`, `-=`, `*=`, `/=` and `%=`, and support the comparison operators. The class contains the following methods:

METHOD	DESCRIPTION
<code>Rep count() const</code>	Returns the duration value as the number of ticks. The return type is the type specified as parameter to the <code>duration</code> template.
<code>static duration zero()</code>	Returns a duration with a duration value equivalent to 0.
<code>static duration min()</code>	Returns a duration with the minimum/maximum possible duration value representable by the type specified as parameter to the <code>duration</code> template.
<code>static duration max()</code>	

Let's see how durations can be used in actual code. A duration where each tick is one second can be defined as follows:

```
duration<long> d1;
```

Because `ratio<1>` is the default tick period, this is the same as writing the following:

```
duration<long, ratio<1>> d1;
```

The following specifies a duration in minutes (tick period = 60 seconds):

```
duration<long, ratio<60>> d2;
```

To define a duration where each tick period is a sixtieth of a second, use the following:

```
duration<double, ratio<1, 60>> d3;
```

As seen earlier in this chapter, the `<ratio>` header file defines a number of SI rational constants. These predefined constants come in handy for defining tick periods. For example, the following line of code defines a duration where each tick period is one millisecond:

```
duration<long long, milli> d4;
```

The following example demonstrates several aspects of durations. It shows you how to define durations, how to perform arithmetic operations on them, and how to convert one duration into another duration with a different tick period:

```
// Specify a duration where each tick is 60 seconds
duration<long, ratio<60>> d1(123);
cout << d1.count() << endl;
// Specify a duration represented by a double with each tick
// equal to 1 second and assign the largest possible duration to it.
duration<double> d2;
d2 = d2.max();
cout << d2.count() << endl;
// Define 2 durations:
// For the first duration, each tick is 1 minute
// For the second duration, each tick is 1 second
duration<long, ratio<60>> d3(10); // = 10 minutes
duration<long, ratio<1>> d4(14); // = 14 seconds
// Compare both durations
if (d3 > d4)
    cout << "d3 > d4" << endl;
else
    cout << "d3 <= d4" << endl;
// Increment d4 with 1 resulting in 15 seconds
++d4;
// Multiply d4 by 2 resulting in 30 seconds
d4 *= 2;
// Add both durations and store as minutes
duration<double, ratio<60>> d5 = d3 + d4;
// Add both durations and store as seconds
duration<long, ratio<1>> d6 = d3 + d4;
cout << d3.count() << " minutes + " << d4.count() << " seconds = "
    << d5.count() << " minutes or "
```

```

    << d6.count() << " seconds" << endl;
// Create a duration of 30 seconds
duration<long> d7(30);
// Convert the seconds of d7 to minutes
duration<double, ratio<60>> d8(d7);
cout << d7.count() << " seconds = " << d8.count() << " minutes" << endl;

```

The output is as follows:

```

123
1.79769e+308
d3 > d4
10 minutes + 30 seconds = 10.5 minutes or 630 seconds
30 seconds = 0.5 minutes

```

NOTE *The second line in the output represents the largest possible duration with type double. The exact value might be different depending on your compiler.*

Pay special attention to the following two lines:

```

duration<double, ratio<60>> d5 = d3 + d4;
duration<long, ratio<1>> d6 = d3 + d4;

```

They both calculate `d3+d4` but the first one stores it as a floating point value representing minutes while the second one stores the result as an integer representing seconds. Conversion from minutes to seconds or vice versa happens automatically.

The following two lines from the preceding example demonstrate how to explicitly convert between different units of time:

```

duration<long> d7(30);           // seconds
duration<double, ratio<60>> d8(d7); // minutes

```

The first line defines a duration representing 30 seconds. The second line converts these 30 seconds into minutes, resulting in 0.5 minutes. Converting in this direction can result in a non-integral value and thus requires you to use a duration represented by a floating point type; otherwise, you will get some cryptic compiler errors. The following lines, for example, will not compile because `d8` is using `long` instead of a floating point type:

```

duration<long> d7(30);           // seconds
duration<long, ratio<60>> d8(d7); // minutes // Error

```

Converting in the other direction does not require floating point types if the source is an integral type, because the result is always an integral value if you started with an integral value. For example, the following lines convert 10 minutes into seconds, both represented by the integral type `long`:

```

duration<long, ratio<60>> d9(10); // minutes
duration<long> d10(d9);           // seconds

```

The library also provides the following standard duration types, where X stands for “signed integer type of at least”:

```
typedef duration<X 64 bits, nano>      nanoseconds;
typedef duration<X 55 bits, micro>      microseconds;
typedef duration<X 45 bits, milli>      milliseconds;
typedef duration<X 35 bits>            seconds;
typedef duration<X 29 bits, ratio<60>>  minutes;
typedef duration<X 23 bits, ratio<3600>> hours;
```

The exact type of X depends on your compiler, but the C++ standard requires it to be a signed integer type of at least the specified size. The preceding `typedefs` make use of the predefined SI `ratio` `typedefs` as described earlier in this chapter. The following is an example of how to use these predefined durations. The code first defines a variable `t`, which is the result of 1 hour + 23 minutes + 45 seconds. The `auto` keyword is used to let the compiler automatically figure out the exact type of `t`. The second line uses the constructor of the predefined `seconds` duration to convert the value of `t` to seconds and writes the result to the console:

```
auto t = hours(1) + minutes(23) + seconds(45);
cout << seconds(t).count() << " seconds" << endl;
```

Because the standard requires that the predefined durations use integer types, there can be compiler errors if a conversion **could** end up with a non-integral value. While integer division normally truncates, in the case of durations, which are implemented by `ratio` types, the compiler declares any computation that could result in a non-zero remainder as a compile-time error. For example, the following will not compile because converting 90 seconds results in 1.5 minutes:

```
seconds s(90);
minutes m(s);
```

However, the following will not compile either, even though 60 seconds is exactly 1 minute. It is flagged as a compile-time error because converting from seconds to minutes could result in non-integral values:

```
seconds s(60);
minutes m(s);
```

Converting in the other direction works perfectly fine because the `minutes` duration is an integral value and converting it to `seconds` always results in an integral value:

```
minutes m(2);
seconds s(m);
```

You can use the standard user-defined literals “`n`”, “`min`”, “`s`”, “`ms`”, “`us`”, and “`ns`” for creating durations. For example:

```
auto myDuration = 42min;    // 42 minutes
```



Clock

A `clock` is a class consisting of a `time_point` and a `duration`. The `time_point` type is discussed in detail in the next section, but those details are not required to understand how `clocks` work. However, `time_point`s themselves depend on `clocks`, so it's important to know the details of `clocks` first.

Three `clocks` are defined by the standard. The first is called `system_clock` and represents the wall clock time from the system-wide realtime clock. The second is called `steady_clock`, a clock that guarantees its `time_point` will never decrease, which is not guaranteed for `system_clock` because the system clock can be adjusted at any time. The third is the `high_resolution_clock`, which has the shortest possible tick period. Depending on your compiler, it is possible for the `high_resolution_clock` to be a synonym for `steady_clock` or `system_clock`.

Every `clock` has a static `now()` method to get the current time as a `time_point`. The `system_clock` also defines two static helper functions for converting `time_point`s to and from the `time_t` C-style time representation. The first is called `to_time_t()` converting a given `time_point` to a `time_t`; the second is called `from_time_t()`, which returns a `time_point` initialized with a given `time_t` value. The `time_t` type is defined in the `<ctime>` header file.

The following example shows a complete program, which gets the current time from the system and outputs the time in a human readable format on the console. The `localtime()` function converts a `time_t` to a local time represented by `tm` and is defined in the `<ctime>` header file. The `put_time()` stream manipulator, defined in the `<iomanip>` header, is introduced in Chapter 12:

```
// Get current time as a time_point
system_clock::time_point tpoint = system_clock::now();
// Convert to a time_t
time_t tt = system_clock::to_time_t(tpoint);
// Convert to local time
tm* t = localtime(&tt);
// Write the time to the console
cout << put_time(t, "%H:%M:%S") << endl;
```

If your compiler does not yet support the `put_time()` manipulator, you can use the C-style `strftime()` function, defined in `<ctime>`, as follows. Using the old `strftime()` function requires you to supply a buffer that is big enough to hold the human readable representation of the given time:

```
// Get current time as a time_point
system_clock::time_point tpoint = system_clock::now();
// Convert to a time_t
time_t tt = system_clock::to_time_t(tpoint);
// Convert to local time
tm* t = localtime(&tt);
// Convert to readable format
char buffer[80] = {0};
strftime(buffer, sizeof(buffer), "%H:%M:%S", t);
// Write the time to the console
cout << buffer << endl;
```

NOTE These examples might give you a security-related error or warning on the call to `localtime()`. With Microsoft Visual C++ you can use the safe version called `localtime_s()`. On Linux you can use `localtime_r()`.

The Chrono library can also be used to time how long it takes for a piece of code to execute. The following example shows how you can do this. The actual type of the variables `start` and `end` is `system_clock::time_point` and the actual type of `diff` is a `duration`:

```
// Get the start time
auto start = system_clock::now();
// Execute code that you want to time
double d = 0;
for (int i = 0; i < 1000000; ++i) {
    d += sqrt(sin(i) * cos(i));
}
// Get the end time and calculate the difference
auto end = system_clock::now();
auto diff = end - start;
// Convert the difference into milliseconds and print on the console
cout << duration<double, milli>(diff).count() << "ms" << endl;
```

The loop in this example is performing some arithmetic operations with `sqrt()`, `sin()`, and `cos()` to make sure the loop doesn't end too fast. If you get really small values for the difference in milliseconds on your system, those values will not be accurate and you should increase the number of iterations of the loop to make it last longer. Small timings will not be accurate, because, while timers often have a resolution in milliseconds, on most operating systems, this timer is updated infrequently, for example, every 10ms or 15ms. This induces a phenomenon called *gating error*, where any event that occurs in less than 1 timer tick appears to take place in zero units of time; any event between 1 and 2 timer ticks appears to take place in 1 timer unit. For example, on a system with a 15ms timer update, a loop that takes 44ms will appear to take only 30ms. When using such timers to time computations, it is important to make sure that the entire computation takes place across a fairly large number of basic timer tick units so that these errors are minimized.

Time Point

A point in time is represented by the `time_point` class and stored as a `duration` relative to the *epoch*. A `time_point` is always associated with a certain `clock` and the epoch is the origin of this associated `clock`. For example, the epoch for the classic Unix/Linux time is 1st of January 1970, and durations are measured in seconds. The epoch for Windows is 1st of January 1601 and durations are measured in 100 nanosecond units. Other operating systems have different epoch dates and duration units.

The `time_point` class contains a function called `time_since_epoch()`, which returns a `duration` representing the time between the epoch of the associated `clock` and the stored point in time. A `time_point` supports arithmetic operations that make sense for time points such as `+`, `-`, `+=` and `-=`. Comparison operators are also supported to compare two time points. Two static methods are provided: `min()` returning the smallest possible point in time, and `max()` returning the largest possible point in time.

The `time_point` class has three constructors:

- `time_point():` constructs a `time_point` initialized with `duration::zero()`. The resulting `time_point` represents the epoch of the associated `clock`.
- `time_point(const duration& d):` constructs a `time_point` initialized with the given duration. The resulting `time_point` represents `epoch + d`.
- `template<class Duration2> time_point(const time_point<clock, Duration2>& t):` constructs a `time_point` initialized with `t.time_since_epoch()`.

Each `time_point` is associated with a `clock`. To create a `time_point`, you specify the `clock` as the template parameter:

```
time_point<steady_clock> tp1;
```

Each `clock` also knows its `time_point` type, so you can also write it as follows:

```
steady_clock::time_point tp1;
```

The following example demonstrates the `time_point` class:

```
// Create a time_point representing the epoch
// of the associated steady clock
time_point<steady_clock> tp1;
// Add 10 minutes to the time_point
tp1 += minutes(10);
// Store the duration between epoch and time_point
auto d1 = tp1.time_since_epoch();
// Convert the duration to seconds and write to console
duration<double> d2(d1);
cout << d2.count() << " seconds" << endl;
```

The output should be:

```
600 seconds
```

RANDOM NUMBER GENERATION

Generating good random numbers in software is a complex topic. Before C++11, the only way to generate random numbers was to use the C-style `rand()` and `rand()` functions. The `rand()` function needs to be called once in your application and is used to initialize the random number generator, also called *seeding*. Usually the current system time would be used as a seed.

WARNING *You need to make sure that you use a good quality seed for your software-based random number generator. If you initialize the random number generator with the same seed every time, you will create the same sequence of random numbers every time. This is why the seed is usually the current system time.*

Once the generator is initialized, random numbers can be generated with `rand()`. The following example shows how to use `srand()`, to initialize the generator with the current system time as the seed, and `rand()`, to generate a random number. The `time(nullptr)` call returns the system time, and is defined in the `<ctime>` header file:

```
 srand(static_cast<unsigned int>(time(nullptr)));
 cout << rand() << endl;
```

A random number within a certain range can be generated with the following function:

```
int getRandom(int min, int max)
{
    return (rand() % static_cast<int>(max + 1 - min)) + min;
}
```

The old C-style `rand()` function generates random numbers in the range 0 to `RAND_MAX`, which is defined by the standard to be at least 32767. Unfortunately, the low-order bits of `rand()` are often not very random, which means, using the previous `getRandom()` function to generate a random number in a small range, such as 1 to 6, will not result in very good randomness.

NOTE *Software-based random number generators can never generate truly random numbers and are therefore called pseudo-random number generators because they rely on mathematical formulas to give the impression of randomness.*

The old `srand()` and `rand()` functions don't offer much in terms of flexibility. You cannot, for example, change the distribution of the generated random numbers. C++11 has added a powerful library to generate random numbers by using different algorithms and distributions. The library is defined in the `<random>` header file. The library has three big components: *engines*, *engine adapters*, and *distributions*. A random number *engine* is responsible for generating the actual random numbers and storing the state for generating subsequent random numbers. The *distribution* determines the range of the generated random numbers and how they are mathematically distributed within that range. A random number *engine adapter* modifies the results of a random number engine you associate it with.

It's highly recommended to stop using `srand()` and `rand()`, and to start using the classes from `<random>`.

Random Number Engines

C++ defines the following random number engine templates:

- `random_device`
- `linear_congruential_engine`
- `mersenne_twister_engine`
- `subtract_with_carry_engine`

The `random_device` engine is not a software-based generator; it is a special engine that requires a piece of hardware attached to your computer that generates truly non-deterministic random numbers, for example by using the laws of physics. A classic mechanism measures the decay of a radioactive isotope by counting alpha-particles-per-time-interval or something like that, but there are many other kinds of physics-based random-number generators, including measuring the “noise” of reverse-biased diodes (thus eliminating the concerns about radioactive sources in your computer). The details of these mechanisms fall outside the scope of this book.

According to the specification for `random_device`, if no such device is attached to the computer, the library is free to use one of the software algorithms. The choice of algorithm is up to the library designer.

The quality of a random number generator is referred to as its *entropy* measure. The `entropy()` method of the `random_device` class returns 0.0 if it is using a software-based pseudo-random number generator, and returns a nonzero value if there is a hardware device attached. The nonzero value is an estimate of the entropy of the attached device.

Using the `random_device` engine is rather straightforward:

```
random_device rnd;
cout << "Entropy: " << rnd.entropy() << endl;
cout << "Min value: " << rnd.min()
     << ", Max value: " << rnd.max() << endl;
cout << "Random number: " << rnd() << endl;
```

A possible output of this program could be as follows:

```
Entropy: 32
Min value: 0, Max value: 4294967295
Random number: 3590924439
```

Next to the `random_device` engine, there are three pseudo-random number engines:

- The *linear congruential engine* requires a minimal amount of memory to store its state. The state is a single integer containing the last generated random number or the initial seed if no random number has been generated yet. The period of this engine depends on an algorithmic parameter and can be up to 2^{64} but usually less. For this reason, the linear congruential engine should not be used when you need a high-quality random number sequence.
- From the three pseudo-random number engines, the *Mersenne twister* generates the highest quality of random numbers. The period of a Mersenne twister depends on an algorithmic parameter but is much bigger than the period of a linear congruential engine. The memory required to store the state of a Mersenne twister also depends on its parameters but is much larger than the single integer state of the linear congruential engine. For example, the predefined Mersenne twister `mt19937` has a period of $2^{19937}-1$, while the state contains 624 integers or around 2.5 kilobytes.
- The *subtract with carry engine* requires a state of 25 integers or around 100 bytes, however, the quality of the generated random numbers is less than the numbers generated by the Mersenne twister.

The mathematical details of the engines fall outside the scope of this book, and defining the quality of random numbers requires a mathematical background. If you want to know more about this topic, you can consult a reference from the “Random Numbers” section in Appendix B.

The `random_device` engine is easy to use and doesn’t require any parameters. However, creating an instance of one of the three pseudo-random number generators requires you to specify a number of mathematical parameters, which can be complicated. The selection of parameters greatly influences the quality of the generated random numbers. For example, the definition of the `mersenne_twister_engine` class looks as follows:

```
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
         UIntType a, size_t u, UIntType d, size_t s,
         UIntType b, size_t t, UIntType c, size_t l, UIntType f>
class mersenne_twister_engine {...}
```

It requires 14 parameters. The `linear_congruential_engine` and the `subtract_with_carry_engine` classes also require a number of these mathematical parameters. For this reason, the standard defines a couple of predefined engines. One example is the `mt19937` Mersenne twister which is defined as follows:

```
typedef mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31,
                           0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18,
                           1812433253> mt19937;
```

These parameters are all magic, unless you understand the details of the Mersenne twister algorithm. In general, you do not want to modify any of these parameters unless you are a specialist in the mathematics of pseudo-random number generators. Instead, it is highly recommended to use one of the predefined `typedefs` such as `mt19937`. A complete list of predefined engines is given in a later section.

Random Number Engine Adapters

A random number engine adapter modifies the result of a random number engine you associate it with, which is called the *base engine*. This is an example of the *adapter pattern*. The following three adapter templates are defined:

```
template<class Engine, size_t p, size_t r> class
    discard_block_engine {...}
template<class Engine, size_t w, class UIntType> class
    independent_bits_engine {...}
template<class Engine, size_t k> class
    shuffle_order_engine {...}
```

The `discard_block_engine` adapter generates random numbers by discarding some of the values generated by its base engine. It requires three parameters: the engine to adapt, the block size p , and the used block size r . The base engine is used to generate p random numbers. The adapter then discards $p-r$ of those numbers and returns the remaining r numbers.

The `independent_bits_engine` adapter generates random numbers with a given number of bits w by combining several random numbers generated by the base engine.

The `shuffle_order_engine` adapter generates the same random numbers that are generated by the base engine, but delivers them in a different order.

The exact working of these adapters depends on mathematics and falls outside the scope of this book.

The standard includes a number of predefined engine adapters. The following section lists the predefined engines and engine adapters.

Predefined Engines and Engine Adapters

As mentioned earlier, it is not recommended to specify your own parameters for pseudo-random number engines or engine adapters, but instead to use one of the standard ones. C++ defines the following predefined engines and engine adapters, all in the `<random>` header file:

```
typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>
    minstd_rand0;
typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>
    minstd_rand;
typedef mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31,
    0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18,
    1812433253> mt19937;
typedef mersenne_twister_engine<uint_fast64_t, 64, 312, 156, 31,
    0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67ffffeda60000, 37,
    0xffff7eee00000000, 43, 6364136223846793005> mt19937_64;
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
    ranlux24_base;
typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12> ranlux48_base;
typedef discard_block_engine<ranlux24_base, 223, 23> ranlux24;
typedef discard_block_engine<ranlux48_base, 389, 11> ranlux48;
typedef shuffle_order_engine<minstd_rand0, 256> knuth_b;
typedef implementation-defined default_random_engine;
```

The `default_random_engine` is compiler dependent.

The following section gives an example of how to use these predefined engines.

Generating Random Numbers

Before you can generate any random number, you first need to create an instance of an engine. If you use a software-based engine, you will also need to define a distribution. A distribution is a mathematical formula describing how numbers are distributed within a certain range. The recommended way to create an engine is to use one of the predefined engines as discussed earlier.

The following example uses the predefined engine called `mt19937`, using a Mersenne twister engine. This is a software-based generator. Just as with the old `srand()`/`rand()` generator, a software-based engine should be initialized with a seed. The seed for `srand()` was often the current time. In modern C++ it's recommended not to use any time-based seeds, but instead to use `random_device` to generate a seed.

```
random_device rndDevice;
mt19937 eng(rndDevice());
```

Next, a distribution is defined. This example uses a uniform integer distribution, for the range 1 to 99. Distributions are explained in detail in the next section, but the uniform distribution is easy enough to use for this example:

```
uniform_int_distribution<int> dist(1, 99);
```

Once the engine and distribution are defined, random numbers can be generated by calling a function whose name is the name of the distribution and passing as a parameter the engine. For this example this is written as `dist(eng)`:

```
cout << dist(eng) << endl;
```

As you can see, to generate a random number using a software-based engine, you always need to specify the engine and distribution. The `std::bind()` utility introduced in Chapter 17 and defined in the `<functional>` header file can be used to remove the need to specify both the distribution and the engine when generating a random number. The following example uses the same `mt19937` engine and uniform distribution as the previous example. It then defines `gen()` by using `std::bind()` to bind `eng` to the first parameter of `dist()`. This way, you can call `gen()` without any argument to generate a new random number. The `auto` keyword is used in the definition of `gen()` to avoid having to write the exact type ourselves. The example then demonstrates the use of `gen()` in combination with the `generate()` algorithm to fill a vector of 10 elements with random numbers. The `generate()` algorithm is discussed in Chapter 17 and is defined in `<algorithm>`:

```
random_device rndDevice;
mt19937 eng(rndDevice());
uniform_int_distribution<int> dist(1, 99);
auto gen = std::bind(dist, eng);
vector<int> vec(10);
generate(begin(vec), end(vec), gen);
for (auto i : vec) {
    cout << i << " ";
}
```

NOTE Remember that the `generate()` algorithm overwrites existing elements and does not insert new elements. This means that you first need to size the vector to hold the number of elements you need, and then call the `generate()` algorithm. The previous example sizes the vector by specifying the size as argument to the constructor.

Even though you don't know the exact type of `gen()`, it's still possible to pass `gen()` to another function that wants to use that generator. However, you cannot use a normal function; you need to use a function template. The previous example can be adapted to do the generation of random numbers in a function called `fillVector()`, which is a function template that looks as follows:

```
template<typename T>
void fillVector(vector<int>& vec, T rndGen)
{
    generate(begin(vec), end(vec), rndGen);
}
```

This function is used as follows:

```
random_device rndDevice;
mt19937 eng(rndDevice());
uniform_int_distribution<int> dist(1, 99);
auto gen = std::bind(dist, eng);
vector<int> vec(10);
fillVector(vec, gen);
for (auto i : vec) {
    cout << i << " ";
}
```

Random Number Distributions

A distribution is a mathematical formula describing how numbers are distributed within a certain range. The random number generator library comes with the following distributions that can be used with pseudo-random number engines to define the distribution of the generated random numbers. It's a compacted representation. The first line of each distribution is the class name and class template parameters, if any. The next lines are a constructor for the distribution. Only one constructor for each distribution is shown to give you an idea of the class. Consult a Standard Library Reference — for example <http://www.cppreference.com/> or <http://www.cplusplus.com/> reference — for a detailed list of all constructors and methods of each distribution.

Uniform distributions:

```
template<class IntType = int> class uniform_int_distribution
uniform_int_distribution(IntType a = 0,
                        IntType b = numeric_limits<IntType>::max());
template<class RealType = double> class uniform_real_distribution
uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
```

Bernoulli distributions:

```
class bernoulli_distribution
bernoulli_distribution(double p = 0.5);
template<class IntType = int> class binomial_distribution
binomial_distribution(IntType t = 1, double p = 0.5);
template<class IntType = int> class geometric_distribution
geometric_distribution(double p = 0.5);
template<class IntType = int> class negative_binomial_distribution
negative_binomial_distribution(IntType k = 1, double p = 0.5);
```

Poisson distributions:

```
template<class IntType = int> class poisson_distribution
poisson_distribution(double mean = 1.0);
template<class RealType = double> class exponential_distribution
exponential_distribution(RealType lambda = 1.0);
template<class RealType = double> class gamma_distribution
gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
template<class RealType = double> class weibull_distribution
weibull_distribution(RealType a = 1.0, RealType b = 1.0);
```

```
template<class RealType = double> class extreme_value_distribution
    extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
```

Normal distributions:

```
template<class RealType = double> class normal_distribution
    normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
template<class RealType = double> class lognormal_distribution
    lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
template<class RealType = double> class chi_squared_distribution
    chi_squared_distribution(RealType n = 1);
template<class RealType = double> class cauchy_distribution
    cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
template<class RealType = double> class fisher_f_distribution
    fisher_f_distribution(RealType m = 1, RealType n = 1);
template<class RealType = double> class student_t_distribution
    student_t_distribution(RealType n = 1);
```

Sampling distributions:

```
template<class IntType = int> class discrete_distribution
    discrete_distribution(initializer_list<double> wl);
template<class RealType = double> class piecewise_constant_distribution
    template<class UnaryOperation>
        piecewise_constant_distribution(initializer_list<RealType> bl,
                                         UnaryOperation fw);
template<class RealType = double> class piecewise_linear_distribution
    template<class UnaryOperation>
        piecewise_linear_distribution(initializer_list<RealType> bl,
                                         UnaryOperation fw);
```

Each distribution requires a set of parameters. Explaining all these mathematical parameters in detail is outside the scope of this book, but the rest of this section gives a couple of examples to explain the impact of a distribution on the generated random numbers.

Distributions are easiest to understand when you look at a graphical representation of them. For example, the following code generates one million random numbers between 1 and 99 and counts how many times a certain number between 1 and 99 is randomly chosen. This is stored in a `map` where the key is a number between 1 and 99, and the value associated with a key is the number of times that that key has been selected randomly. After the loop, the results are written to a CSV (Comma Separated Values) file, which can be opened in a spreadsheet application to generate a graphical representation:

```
const unsigned int kStart = 1;
const unsigned int kEnd = 99;
const unsigned int kIterations = 1000000;
// Uniform Mersenne Twister
random_device rndDevice;
mt19937 eng(rndDevice());
uniform_int_distribution<int> dist(kStart, kEnd);
auto gen = bind(dist, eng);
```

```

map<int, int> m;
for (unsigned int i = 0; i < kIterations; ++i) {
    int rnd = gen();
    // Search map for a key = rnd. If found, add 1 to the value associated
    // with that key. If not found, add the key to the map with value 1.
    ++(m[rnd]);
}
// Write to a CSV file
ofstream of("res.csv");
for (unsigned int i = kStart; i <= kEnd; ++i) {
    of << i << ",";
    auto res = m.find(i);
    of << (res != end(m) ? res->second : 0) << endl;
}

```

The resulting data can then be used to generate a graphical representation. The graph of the preceding uniform Mersenne twister is shown in Figure 19-1.

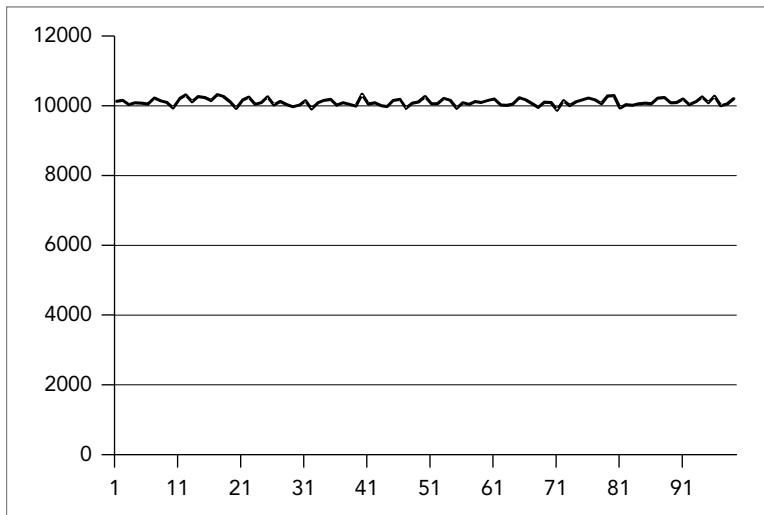


FIGURE 19-1

The horizontal axis represents the range in which random numbers are generated. The graph clearly shows that all numbers between 1 and 99 are randomly chosen around 10,000 times and that the distribution of the generated random numbers is uniform across the entire range.

The example can be modified to generate random numbers according to a normal distribution instead of a uniform distribution. Only two small changes are required. First, the creation of the distribution is modified as follows:

```
normal_distribution<double> dist(50, 10);
```

Because normal distributions use doubles instead of integers, you also need to modify the call to `gen()`:

```
int rnd = static_cast<int>(gen());
```

Figure 19-2 shows a graphical representation of the random numbers generated according to the normal distribution.

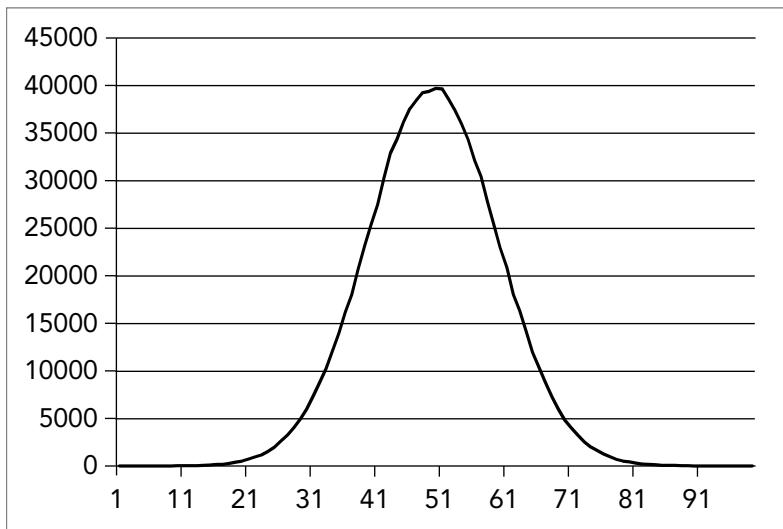


FIGURE 19-2

The graph clearly shows that most of the generated random numbers will be around the center of the range. In this example, the value 50 will be randomly chosen around 40,000 times, while values like 20 or 80 will be chosen randomly only around 500 times.

TUPLES

The `std::pair` class, defined in `<utility>` and introduced in Chapter 16, can store two values, each with a specific type. The type of each value should be known at compile time. The following is a short example:

```
pair<int, string> p1(16, "Hello World");
pair<bool, float> p2(true, 0.123f);
cout << "p1 = (" << p1.first << ", " << p1.second << ")" << endl;
cout << "p2 = (" << p2.first << ", " << p2.second << ")" << endl;
```

The output is as follows:

```
p1 = (16, Hello World)
p2 = (1, 0.123)
```

C++ also has a `std::tuple` class, defined in the `<tuple>` header file. A `tuple` is a generalization of a `pair`, and allows you to store any number of values, each with its own specific type. Just like a `pair`, a `tuple` has a fixed size and fixed value types, determined at compile time.

A `tuple` can be created with a `tuple` constructor, specifying the template types and specifying the actual values. For example, the following code creates a `tuple` where the first element is an integer, the second element a `string`, and the last element a Boolean:

```
using MyTuple = tuple<int, string, bool>;
MyTuple t1(16, "Test", true);
```

`std::get<i>()` is used to get the i -th element from a `tuple`, where i is a 0-based index; that is `<0>` is the first element of the tuple, `<1>` is the second element of the tuple and so on. The value returned has the correct type for that index in the `tuple`:

```
cout << "t1 = (" << get<0>(t1) << ", " << get<1>(t1)
     << ", " << get<2>(t1) << ")" << endl;
// Outputs: t1 = (16, Test, 1)
```

You can check that `get<i>()` returns the correct type by using `typeid()`, from the `<typeinfo>` header. The output of the following code says that the value returned by `get<1>(t1)` is indeed a `std::string`:

```
cout << "Type of get<1>(t1) = " << typeid(get<1>(t1)).name() << endl;
// Outputs: Type of get<1>(t1) = class std::basic_string<char,
//           struct std::char_traits<char>, class std::allocator<char> >
```

NOTE *The exact string returned by `typeid()` is compiler dependent. The preceding output is from Visual C++ 2013.*

C++14

C++14 adds the ability to retrieve an element from a `tuple` based on the type with `std::get<T>()` where `T` is the type of the element you want to retrieve instead of the index. The compiler generates an error if the `tuple` has several elements with the requested type. For example, you can retrieve the `string` element from `t1` as follows:

```
cout << "String = " << get<string>(t1) << endl;
// Outputs: String = Test
```

The size of a `tuple` can be queried with the `std::tuple_size` template. Note that `tuple_size` requires you to specify the type of the `tuple` (`MyTuple` in this case) and not an actual `tuple` instance like `t1`:

```
cout << "Tuple Size = " << tuple_size<MyTuple>::value << endl;
// Outputs: Tuple Size = 3
```

Another way to create a `tuple` is to use the `std::make_tuple` utility function. This function only needs the actual values and deduces the types automatically at compile time. Because of the

automatic deduction of types, you cannot use `&` to specify a reference. If you want to use `make_tuple()` to generate a tuple containing a reference or a constant reference, then you need to use `ref()` or `cref()`, respectively, as is demonstrated in the following example. The `ref()` and `cref()` helpers are defined in the `<functional>` header file. For example, the following `make_tuple()` call results in a tuple of type `tuple<int, double&, string&, const string&>`:

```
double d = 3.14;
string str1 = "Test";
auto t2 = make_tuple(16, ref(d), ref(str1), cref(str1));
```

To test the double reference in the `t2` tuple, the following three lines of code first write the value of the `double` variable to the console. It then uses `get<1>(t2)`, which actually returns a reference to `d` because `ref(d)` was used for the second tuple element. The second line changes the value of the variable referenced, and the last line shows that the value of `d` is indeed changed through the reference stored in the tuple:

```
cout << "d = " << d << endl;
get<1>(t2) *= 2;
cout << "d = " << d << endl;
// Outputs: d = 3.14
//           d = 6.28
```

The following block of code demonstrates something similar, but uses `strings`. It tries to change the value of the variable referenced by the third element of the tuple, which works perfectly because of the use of `ref()`, which results in a `string&`. However, trying to change the variable referenced by the fourth element will fail because that is a `const string&`:

```
cout << "str = " << str1 << endl;
get<2>(t2) = "Hello";
//get<3>(t2) = "Hello";      // ERROR because of cref()
cout << "str = " << str1 << endl;
// Outputs: str = Test
//           str = Hello
```

Tuples can also contain other containers. The following line creates a tuple where the first element is a `string` and the second element is a `vector` of integers:

```
tuple<string, vector<int>> t1("test", {1,2,3,4});
```

The standard defines a `std::tie()` utility function, which generates a tuple of references. The following example first creates a tuple consisting of an integer, a `string`, and a Boolean value. It then creates three variables: an integer, a `string`, and a Boolean and writes the values of those variables to the console. The `tie(i, str, b)` call creates a tuple containing a reference to `i`, a reference to `str`, and a reference to `b`. The assignment operator is used to assign tuple `t1` to the result of `tie()`. Because the result of `tie()` is a tuple of references, the assignment actually changes the values in the three separate variables as is shown by the output of the values after the assignment:

```
tuple<int, string, bool> t1(16, "Test", true);
int i = 0;
string str;
```

```
bool b = false;
cout << "Before: i = " << i << ", str = \" " << str << "\", b = " << b << endl;
tie(i, str, b) = t1;
cout << "After: i = " << i << ", str = \" " << str << "\", b = " << b << endl;
```

The result is as follows:

```
Before: i = 0, str = "", b = 0
After: i = 16, str = "Test", b = 1
```

You can use `std::tuple_cat()` to concatenate two tuples into one tuple. In the following example, the type of `t3` will be `tuple<int, string, bool, double, string>`:

```
tuple<int, string, bool> t1(16, "Test", true);
tuple<double, string> t2(3.14, "string 2");
auto t3 = tuple_cat(t1, t2);
```

Tuples also support the following comparison operators: `==`, `<`, `!=`, `>`, `<=` and `>=`. For the comparison operators to work, the element types stored in the tuple should support them as well. For example:

```
tuple<int, string> t1(123, "def");
tuple<int, string> t2(123, "abc");
if (t1 < t2) {
    cout << "t1 < t2" << endl;
} else {
    cout << "t1 >= t2" << endl;
}
```

The output should be as follows:

```
t1 >= t2
```

Iterating over the values of a tuple is unfortunately not straightforward. You cannot write a simple loop and call something like `get<i>(mytuple)` because the value of `i` must be known at compile time. A solution is to use template metaprogramming, which is discussed in detail in Chapter 21, together with an example on how to print tuple values.

SUMMARY

This chapter gave an overview of additional functionality provided by the C++ standard that did not fit in other chapters. You learned how to use `std::function` to create function pointers. This chapter also gave an overview of the `ratio` template to define compile-time rational numbers, the Chrono library, and the random number generation library. The chapter finished with a discussion on tuples, which are a generalization of pairs.

This chapter concludes part 3 of the book. The next part discusses some more advanced topics and starts with a chapter showing you how to customize and extend the functionality provided by the C++ Standard Template Library.

PART IV

Mastering Advanced Features of C++

- **CHAPTER 20:** Customizing and Extending the STL
- **CHAPTER 21:** Advanced Templates
- **CHAPTER 22:** Memory Management
- **CHAPTER 23:** Multithreaded Programming with C++

20

Customizing and Extending the STL

WHAT'S IN THIS CHAPTER?

- Explaining allocators
- Explaining iterator adapters and how to use the standard iterator adapters
- How to extend the STL
- How to write your own algorithms
- How to write your own containers
- How to write your own iterators

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

The previous chapters show that the STL is a powerful general-purpose collection of containers and algorithms. The information covered so far should be sufficient for most applications. However, those chapters show only the basic functionality of the library. The STL can be customized and extended however you like. For example, you can apply iterators to input and output streams; write your own containers, algorithms, and iterators; and even specify your own memory allocation schemes for containers to use. This chapter provides a taste of these advanced features, primarily through the development of a new STL container: the `hash_map`.

NOTE *Customizing and extending the STL is rarely necessary. If you’re happy with the STL containers and algorithms from the previous chapters, you can skip this one. However, if you really want to understand the STL, not just use it, give this chapter a chance. You should be comfortable with the operator-overloading material of Chapter 14, and because this chapter uses templates extensively, you should also be comfortable with the template material in Chapter 11 before continuing!*

ALLOCATORS

Every STL container takes an `Allocator` type as a template parameter, for which the default usually suffices. For example, the `vector` template definition looks like this:

```
template <class T, class Allocator = allocator<T>> class vector;
```

The container constructors then allow you to specify an object of type `Allocator`. These extra parameters permit you to customize the way the containers allocate memory. Every memory allocation performed by a container is made with a call to the `allocate()` method of the `Allocator` object. Conversely, every deallocation is performed with a call to the `deallocate()` method of the `Allocator` object. The standard library provides a default `Allocator` class called `allocator`, which implements these methods as wrappers for `operator new` and `operator delete`.

If you want containers in your program to use a custom memory allocation and deallocation scheme, you can write your own `Allocator` class. There are several reasons for using custom allocators. For example, if the underlying allocator has unacceptable performance, there are alternatives that can be constructed. Or, if memory fragmentation is a problem (lots of different allocations and deallocations leaving unusable small holes in memory), a single “pool” of objects of one type can be created, called a memory pool. When OS-specific capabilities, such as shared memory segments, must be allocated, using custom allocators allows the use of STL containers in those shared memory segments. The use of custom allocators is complex, and there are many possible problems if you are not careful, so this should not be approached lightly.

Any class that provides `allocate()`, `deallocate()`, and several other required methods and `typedefs` can be used in place of the default `allocator` class. However, in my experience, this feature is rarely used. I’ve never used it myself, so details are omitted from this book. For more details, consult one of the books on the C++ Standard Library listed in Appendix B.

ITERATOR ADAPTERS

The Standard Library provides four *iterator adapters*: special iterators built on top of other iterators. All four iterator adapters are defined in the `<iterator>` header.

NOTE It's also possible to write your own iterator adapters, but this is not covered in this book. Consult one of the books on the Standard Library listed in Appendix B for details.

Reverse Iterators

The STL provides a `reverse_iterator` class that iterates through a bidirectional or random-access iterator in a reverse direction. Every reversible container in the STL, which happens to be every container that's part of the standard except `forward_list` and the unordered associative containers, supplies a `typedef reverse_iterator` and methods called `rbegin()` and `rend()`. The method `rbegin()` returns a `reverse_iterator` pointing to the last element of the container, and `rend()` returns a `reverse_iterator` pointing to the element before the first element of the container. Applying `operator++` to a `reverse_iterator` calls `operator--` on the underlying container iterator, and vice versa. For example, iterating over a collection from the beginning to the end can be done as follows:

```
for (auto iter = begin(collection); iter != end(collection); ++iter) {}
```

Iterating over the elements in the collection from the end to the beginning can be done using a `reverse_iterator` by calling `rbegin()` and `rend()`. Note that you still call `++iter`:

```
for (auto iter = rbegin(collection); iter != rend(collection); ++iter) {}
```

The `reverse_iterator` is useful mostly with algorithms in the STL that have no equivalents that work in reverse order. For example, the basic `find()` algorithm searches for the first element in a sequence. If you want to find the last element in the sequence, you can use a `reverse_iterator` instead. Note that when you call an algorithm such as `find()` with a `reverse_iterator`, it returns a `reverse_iterator` as well. You can always obtain the underlying iterator from a `reverse_iterator` by calling the `base()` method on the `reverse_iterator`. However, due to the implementation details of `reverse_iterator`, the iterator returned from `base()` always refers to one element past the element referred to by the `reverse_iterator` on which it's called.

Here is an example of `find()` with a `reverse_iterator`:

```
// The implementation of populateContainer() is identical to that shown in
// Chapter 17, so it is omitted here.
vector<int> myVector;
populateContainer(myVector);
int num;
cout << "Enter a number to find: ";
cin >> num;
auto it1 = find(begin(myVector), end(myVector), num);
auto it2 = find(rbegin(myVector), rend(myVector), num);
if (it1 != end(myVector)) {
    cout << "Found " << num << " at position " << it1 - begin(myVector)
        << " going forward." << endl;
    cout << "Found " << num << " at position "
        << it2.base() - 1 - begin(myVector) << " going backward." << endl;
} else {
    cout << "Failed to find " << num << endl;
}
```

One line in this program needs further explanation. The code to print out the position found by the reverse iterator looks like this:

```
cout << "Found " << num << " at position "
<< it2.base() - 1 - begin(myVector) << " going backward." << endl;
```

As noted earlier, `base()` returns an iterator referring to one past the element referred to by the `reverse_iterator`. In order to get to the same element, you must subtract one.

A possible output of this program is as follows:

```
Enter a number (0 to quit): 11
Enter a number (0 to quit): 22
Enter a number (0 to quit): 33
Enter a number (0 to quit): 22
Enter a number (0 to quit): 11
Enter a number (0 to quit): 0
Enter a number to find: 22
Found 22 at position 1 going forward.
Found 22 at position 3 going backward.
```

Stream Iterators

The STL provides adapters that allow you to treat input and output streams as input and output iterators. Using these iterators, you can adapt input and output streams so that they can serve as sources and destinations, respectively, in the various STL algorithms. The `ostream_iterator` class is an *output stream iterator*. It is a template class that takes the element type as a type parameter. Its constructor takes an output stream and a `string` to write to the stream following each element. The `ostream_iterator` class writes elements using operator`<<`.

For example, you can use the `ostream_iterator` with the `copy()` algorithm to print the elements of a container with only one line of code. The first parameter of `copy()` is the start iterator of the range to copy, the second parameter is the end iterator of the range, and the third parameter is the destination iterator:

```
vector<int> myVector(10);
iota(begin(myVector), end(myVector), 1);    // Fill vector with 1,2,3...10
// Print the contents of the vector.
copy(cbegin(myVector), cend(myVector), ostream_iterator<int>(cout, " "));
```

Similarly, you can use the *input stream iterator*, `istream_iterator`, to read values from an input stream using the iterator abstraction. Elements are read using operator`>>`. An `istream_iterator` can be used as sources in the algorithms and container methods. For example, the following piece of code reads integers from the console until the end-of-stream is reached. On Windows this happens when you press `Ctrl+Z` followed by `Enter`. The `accumulate()` algorithm is used to sum all the integers together.

```
cout << "Enter numbers separated by white space." << endl;
cout << "Press Ctrl+Z followed by Enter to stop." << endl;
istream_iterator<int> numbersIter(cin);
istream_iterator<int> endIter;
```

```
int sum = accumulate(numbersIter, endIter, 0);
cout << "Sum: " << sum << endl;
```

Take a moment to reflect on this code. If you remove all the output statements and the variable declarations, the only line left is the call to `accumulate()`. Thanks to the input stream iterators, this single line of code is reading any number of integers from the console and sums them together.

Insert Iterators

As Chapter 17 mentions, algorithms like `copy()` don't insert elements into a container; they simply replace old elements in a range with new ones. In order to make algorithms like `copy()` more useful, the STL provides three *insert iterator adapters* that actually insert elements into a container. They are templated on a container type, and take the actual container reference in their constructor. By supplying the necessary iterator interfaces, these adapters can be used as the destination iterators of algorithms like `copy()`. However, instead of replacing elements in the container, they make calls on their container to actually insert new elements.

The basic `insert_iterator` calls `insert(position, element)` on the container, the `back_insert_iterator` calls `push_back(element)`, and the `front_insert_iterator` calls `push_front(element)`.

For example, you can use the `back_insert_iterator` with the `remove_copy_if()` algorithm to populate `vectorTwo` with all elements from `vectorOne` that are not equal to 100:

```
vector<int> vectorOne, vectorTwo;
populateContainer(vectorOne);
back_insert_iterator<vector<int>> inserter(vectorTwo);
remove_copy_if(cbegin(vectorOne), cend(vectorOne), inserter,
    [](int i){ return i == 100; });
copy(cbegin(vectorTwo), cend(vectorTwo), ostream_iterator<int>(cout, " "));
```

As you can see, when you use insert iterators, you don't need to size the destination containers ahead of time.

You can also use the `back_inserter()` utility function to create a `back_insert_iterator`. For example, in the previous example, you can remove the line that defines the `inserter` variable and rewrite the `remove_copy_if()` call as follows. The result is exactly the same as the previous implementation:

```
remove_copy_if(cbegin(vectorOne), cend(vectorOne),
    back_inserter(vectorTwo), [](int i){ return i == 100; });
```

The `front_insert_iterator` and `insert_iterator` work similarly, except that the `insert_iterator` also takes an initial iterator position in its constructor, which it passes to the first call to `insert(position, element)`. Subsequent iterator position hints are generated based on the return value from each `insert()` call.

One huge benefit of `insert_iterator` is that it allows you to use associative containers as destinations of the modifying algorithms. Chapter 17 explains that the problem with associative containers is that you are not allowed to modify the elements over which you iterate. By using an `insert_iterator`, you can instead insert elements, allowing the container to sort them properly internally. Associative containers actually support a form of `insert()` that takes an iterator position, and are supposed to use the position

as a “hint,” which they can ignore. When you use an `insert_iterator` on an associative container, you can pass the `begin()` or `end()` iterator of the container to use as the hint. Here is the previous example modified so that the destination container is a `set` instead of a `vector`:

```
vector<int> vectorOne;
set<int> setOne;
populateContainer(vectorOne);
insert_iterator<set<int>> inserter(setOne, begin(setOne));
remove_copy_if(cbegin(vectorOne), cend(vectorOne), inserter,
[] (int i) { return i == 100; });
copy(cbegin(setOne), cend(setOne), ostream_iterator<int>(cout, " "));
```

Note that the `insert_iterator` modifies the iterator position hint that it passes to `insert()` after each call to `insert()`, such that the position is one past the just-inserted element.

Move Iterators

Chapter 10 discusses *move semantics*, which can be used to prevent unnecessary copying in cases where you know that the source object will be destroyed after an assignment or copy construction. There is an iterator adapter called `move_iterator`. The dereferencing operator for a `move_iterator` automatically converts the value to an *rvalue reference*, which means that the value can be moved to a new destination without the overhead of copying. Before you can use move semantics, you need to make sure your objects are supporting it. The following `MoveableClass` supports move semantics. For more details, see Chapter 10.

```
class MoveableClass
{
public:
    MoveableClass() {
        cout << "Default constructor" << endl;
    }
    MoveableClass(const MoveableClass& src) {
        cout << "Copy constructor" << endl;
    }
    MoveableClass(MoveableClass&& src) noexcept {
        cout << "Move constructor" << endl;
    }
    MoveableClass& operator=(const MoveableClass& rhs) {
        cout << "Copy assignment operator" << endl;
        return *this;
    }
    MoveableClass& operator=(MoveableClass&& rhs) noexcept {
        cout << "Move assignment operator" << endl;
        return *this;
    }
};
```

The constructors and assignment operators are not doing anything useful here, except printing a message to make it easy to see which one is being called. Now that you have this class, you can define a `vector` and store a few `MoveableClass` instances in it as follows:

```

vector<MoveableClass> vecSource;
MoveableClass mc;
vecSource.push_back(mc);
vecSource.push_back(mc);

```

The second line of the code creates a `MoveableClass` instance by using the default constructor. The first `push_back()` call triggers the copy constructor to copy `mc` into the vector. After this operation, the vector will have space for one element, the first copy of `mc`. Note that this discussion is based on the growth strategy and the initial size of a `vector` as implemented by Microsoft Visual C++. The C++ standard does not specify the initial capacity of a `vector` nor its growth strategy, so the output can be different on different compilers.

The second `push_back()` call triggers the `vector` to resize itself, to allocate space for the second element. This resizing causes the move constructor to be called to move every element from the old `vector` to the new resized `vector`. After that, the copy constructor is triggered to copy `mc` a second time into the vector. The output is as follows:

```

Default constructor
Copy constructor
Move constructor
Copy constructor

```

You can create a new `vector` called `vecOne` that contains a copy of the elements from `vecSource` as follows:

```
vector<MoveableClass> vecOne(cbegin(vecSource), cend(vecSource));
```

Without using `move_iterators`, this code triggers the copy constructor two times, once for every element in `vecSource`:

```

Copy constructor
Copy constructor

```

By using the `make_move_iterator()` function to create `move_iterators`, the move constructor of `MoveableClass` is called instead of the copy constructor:

```
vector<MoveableClass> vecTwo(make_move_iterator(begin(vecSource)),
                             make_move_iterator(end(vecSource)));
```

This generates the following output:

```

Move constructor
Move constructor

```

WARNING *Once objects have been moved, you should not access the original objects anymore. For example, in the previous example, you should not access objects in `vecSource` after having moved them to `vecTwo`. See Chapter 10 for details on move semantics.*

EXTENDING THE STL

The STL includes many useful containers, algorithms, and iterators that you can use in your applications. It is impossible, however, for any library to include all possible utilities that all potential clients might need. Thus, the best libraries are extensible: they allow clients to adapt and add to the basic capabilities to obtain exactly the functionality they require. The STL is inherently extensible because of its fundamental structure of separating data from the algorithms that operate on them. You can write your own container that can work with the STL algorithms by providing an iterator that conforms to the STL standard. Similarly, you can write a function that works with iterators from the standard containers. Note that you are not allowed to put your own containers and algorithms in the `std` namespace.

Why Extend the STL?

If you sit down to write an algorithm or container in C++, you can either make it adhere to the STL conventions or not. For simple containers and algorithms, it might not be worth the extra effort to follow the STL guidelines. However, for substantial code that you plan to reuse, the effort pays off. First, the code will be easier for other C++ programmers to understand, because you follow well-established interface guidelines. Second, you will be able to use your container or algorithm on the other parts of the STL (algorithms or containers) without needing to provide special hacks or adapters. Finally, it will force you to employ the necessary rigor required to develop solid code.

Writing an STL Algorithm

Chapter 17 describes a useful set of algorithms that are part of the STL, but you will inevitably encounter situations in your programs for which you need new algorithms. When that happens, it is usually not difficult to write your own algorithm that works with STL iterators just like the standard algorithms.

find all()

Suppose that you want to find all the elements matching a predicate in a given range. The `find()` and `find_if()` algorithms are the most likely candidate algorithms, but each returns an iterator referring to only one element. In fact, there is no standard algorithm to find all the elements matching a predicate, but you can write your own version of this functionality called `find_all()`.

The first task is to define the function prototype. You can follow the model established by `copy_if()`. It will be a templated function on three type parameters: the input iterator, the output iterator, and the predicate. Its arguments will be start and end iterators of the input sequence, start iterator of the output sequence, and a predicate object. Just as `copy_if()`, the algorithm returns an iterator into the output sequence that is one-past-the-last element stored in the output sequence. Here is the prototype:

Another option would be to omit the output iterator, and to return an iterator into the input sequence that iterates over all the matching elements in the input sequence. This would require you to write your own iterator class.

The next task is to write the implementation. The `find_all()` algorithm iterates over all elements in the input sequence, calls the predicate on each element, and stores iterators of matching elements in the output sequence. Here is the implementation:

```
template <typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator find_all(InputIterator first, InputIterator last,
                        OutputIterator dest, Predicate pred)
{
    while (first != last) {
        if (pred(*first)) {
            *dest = first;
            ++dest;
        }
        ++first;
    }
    return dest;
}
```

Similar to `copy_if()` the algorithm only overwrites existing elements in the output sequence, so make sure the output sequence is large enough to hold the result, or use an iterator adapter such as `back_inserter` as demonstrated in the following code. After finding all matching elements, the code counts the number of elements found, which is the number of iterators in `matches`. Then, it iterates through the result, printing each element:

```
vector<int> vec{ 3, 4, 5, 4, 5, 6, 5, 8 };
vector<vector<int>::iterator> matches;
find_all(begin(vec), end(vec), back_inserter(matches),
         [] (int i){ return i == 5; });
cout << "Found " << matches.size() << " matching elements: " << endl;
for (auto it : matches) {
    cout << *it << " at position " << (it - cbegin(vec)) << endl;
}
```

The output is as follows:

```
Found 3 matching elements:
5 at position 2
5 at position 4
5 at position 6
```

Iterator Traits

Some algorithm implementations need additional information about their iterators. For example, they might need to know the type of the elements referred to by the iterator in order to store temporary values, or perhaps they want to know whether the iterator is bidirectional or random access.

C++ provides a class template called `iterator_traits` that allows you to find this info. You instantiate the `iterator_traits` class template with the iterator type of interest, and access one

of five `typedefs`: `value_type`, `difference_type`, `iterator_category`, `pointer`, and `reference`. For example, the following function template declares a temporary variable of the type to which an iterator of type `IteratorType` refers. Note the use of the `typename` keyword in front of the `iterator_traits` line. You must specify `typename` explicitly whenever you access a type based on one or more template parameters. In this case, the template parameter `IteratorType` is used to access the `value_type` type:

```
#include <iterator>
template <typename IteratorType>
void iteratorTraitsTest(IteratorType it)
{
    typename std::iterator_traits<IteratorType>::value_type temp;
    temp = *it;
    cout << temp << endl;
}
```

This function can be tested with the following test code:

```
vector<int> v{ 5 };
iteratorTraitsTest(cbegin(v));
```

With this test code, the variable `temp` in the `iteratorTraitsTest()` function will be of type `int`. The output is as follows:

```
5
```

In this example, the `auto` keyword could be used to simplify the code, but that wouldn't show you how to use the `iterator_traits` template.

Writing an STL Container

The C++ standard contains a list of requirements that any container must fulfill in order to qualify as an STL container.

Additionally, if you want your container to be sequential (like a `vector`), associative (like a `map`), or unordered associative (like an `unordered_map`), it must conform to supplementary requirements.

My suggestion when writing a new container is to write the basic container first following the general STL rules such as making it a class template, but without worrying too much about the specific details of STL conformity. After you've developed the basic implementation, you can add the iterator and methods so that it can work with the STL framework. This chapter takes that approach to develop a *hash map*.

A Basic Hash Map

C++11 added support for *unordered associative containers*, also called *hash tables*. These are discussed in Chapter 16. However, pre-C++11 did not include hash tables. Unlike the STL `map` and `set`, which provide logarithmic insertion, lookup, and deletion times, a hash table provides constant time insertion, deletion, and lookup in the average case, linear in the worst case. Instead of storing elements in sorted order, it *hashes*, or maps, each element to a particular *bucket*. As long as the number of elements stored isn't significantly greater than the number of buckets, and the *hash*

function distributes the elements uniformly between the buckets, the insertion, deletion, and lookup operations all run in constant time.

NOTE *This section assumes that you are familiar with hashed data structures. If you are not, consult Chapter 16, which includes a discussion on hash tables, or one of the standard data structure texts listed in Appendix B.*

This section implements a simple, but fully functional, `hash_map`. Like a `map`, a `hash_map` stores key/value pairs. In fact, the operations it provides are almost identical to those provided by the `map`, but with different performance characteristics.

This `hash_map` implementation uses chained hashing (also called open hashing) and does not attempt to provide advanced features such as rehashing. Chapter 16 explains the concept of chained hashing in the section on unordered associative containers.

NOTE *It is recommended to use the standard C++ unordered associative containers, also called hash tables, instead of implementing your own. These unordered associative containers, explained in Chapter 16, are called `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. The `hash_map` in this chapter is used to demonstrate writing STL containers.*

The Hash Function

The first choice when writing a `hash_map` is how to handle hash functions. Recalling the adage that a good abstraction makes the easy case easy and the hard case possible, a good `hash_map` interface allows clients to specify their own hash function and number of buckets in order to customize the hashing behavior for their particular workload. On the other hand, clients that do not have the desire, or ability, to write a good hash function and choose a number of buckets should still be able to use the container without doing so. One solution is to allow clients to provide a hash function and number of buckets in the `hash_map` constructor, but also to provide default values. In this implementation, the hash function is a simple function object containing just a single function call operator. The function object is templated on the key type that it hashes in order to support a templated `hash_map` container. Template specialization can be used to write custom hash functions for certain types. The basic function object looks as follows. Note that everything for the `hash_map` implementation is inside a `ProCpp` namespace so that names don't clash with already existing names:

```
template <typename T>
class hash
{
public:
    size_t operator()(const T& key) const;
};
```

The implementation of the function call operator is tricky because it must apply to keys of any type. The following implementation computes an integer-sized hash value by simply treating the key as a sequence of bytes:

```
// Calculate a hash by treating the key as a sequence
// of bytes and summing the ASCII values of the bytes.
template <typename T>
size_t hash<T>::operator()(const T& key) const
{
    size_t bytes = sizeof(key);
    size_t res = 0;
    for (size_t i = 0; i < bytes; ++i) {
        unsigned char b = *((unsigned char*)&key + i);
        res += b;
    }
    return res;
}
```

Unfortunately, when using this hashing method on `strings`, the function calculates the hash of the entire `string` object, and not just of the actual text. The actual text is probably on the heap, and the `string` object only contains a length and a pointer to the text on the heap. The pointer will be different, even if the text it refers to is the same. The result is that two `string` objects with the same text will generate different hash values. Therefore, it's a good idea to provide a specialization of the hash template for `strings`, and in general for any class that contains dynamically allocated memory. Template specialization is discussed in detail in Chapter 11:

```
// A hash specialization for strings
template <>
class hash<std::string>
{
public:
    size_t operator()(const std::string& key) const;
};

// Calculate a hash by summing the ASCII values of all characters.
size_t hash<std::string>::operator()(const std::string& key) const
{
    size_t sum = 0;
    for (size_t i = 0; i < key.size(); ++i) {
        sum += (unsigned char)key[i];
    }
    return sum;
}
```

If you want to use other pointer types or objects as the key, you should write your own hash specialization for those types.

WARNING *The hash functions shown in this section are examples for a basic `hash_map` implementation. They do not guarantee uniform hashing for all key universes. If you need more mathematically rigorous hash functions, or if you don't know what "uniform hashing" is, consult an algorithm's reference from Appendix B.*

The Hash Map Interface

A `hash_map` supports three basic operations: insertion, deletion, and lookup. Of course, it provides a constructor as well. The implementation discussed in this text omits the copy and move constructor, and the copy and move assignment operators. They are required if you want to copy, move, or assign `hash_maps`. The downloadable source code contains an implementation of those constructors and operators, in case you are interested. Here is the public portion of the `hash_map` class template:

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = hash<Key>>
class hash_map
{
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>

    virtual ~hash_map(); // Virtual destructor

    // Throws invalid_argument if the number of buckets is illegal.
    explicit hash_map(const Compare& comp = Compare(),
                      size_t numBuckets = 101, const Hash& hash = Hash());

    // Inserts the key/value pair x.
    void insert(const value_type& x);

    // Removes the element with key k, if it exists.
    void erase(const key_type& k);

    // Find returns a pointer to the element with key k.
    // Returns nullptr if no element with that key exists.
    value_type* find(const key_type& k);
    const value_type* find(const key_type& k) const;

    // operator[] finds the element with key k or inserts an
    // element with that key if none exists yet. Returns a reference to
    // the value corresponding to that key.
    T& operator[] (const key_type& k);

private:
    // Implementation details not shown yet
};
```

As you can see, the key and value types are both template arguments, like for the STL `map`. A `hash_map` stores `pair<const Key, T>` as the actual elements in the container. The `insert()`, `erase()`, `find()`, and `operator[]` methods are straightforward. However, a few aspects of this interface require further explanation.

The Template Argument `Compare`

Like a `map`, `set`, and other standard containers, a `hash_map` allows the client to specify the comparison type as a template parameter and to pass a specific comparison object of that type in the constructor. Unlike a `map` and `set`, a `hash_map` does not sort elements by key, but must still compare keys for equality. Thus, instead of using `less` as the default comparison, it uses `equal_to`. The comparison object is used only to detect attempts to insert duplicate keys into the container.

The Template Argument Hash

You should be able to change the hashing function to make it better suit the type of elements you want to store in the hash map.

Thus, the `hash_map` template takes four template parameters: the key type, the value type, the comparison type, and the hash type.

The Type Aliases

The `hash_map` class template defines three type aliases:

```
using key_type = Key;
using mapped_type = T;
using value_type = std::pair<const Key, T>;
```

The `value_type`, in particular, is useful for referring to the more cumbersome `pair<const Key, T>`. As you will see, these type aliases are also required for STL containers by the standard.

The Implementation

After you finalize the `hash_map` interface, you need to choose the implementation model. The basic hash table structure generally consists of a fixed number of buckets, each of which can store one or more elements. The buckets should be accessible in constant time based on a `bucket_id` (the result of hashing a key). Thus, a `vector` is the most appropriate container for the buckets. Each bucket must store a list of elements, so the STL `list` can be used as the bucket type. Thus, the final structure is a `vector` of `lists` of `pair<const Key, T>` elements. Here are the private members of the `hash_map` class:

```
private:
    using ListType = std::list<value_type>;
    std::vector<ListType> mBuckets;
    size_t mSize;
    Compare mComp;
    Hash mHash;
```

Without the type aliases for `value_type` and `ListType`, the line declaring `mBuckets` would look like this:

```
std::vector<std::list<std::pair<const Key, T>>> mBuckets;
```

The `mComp` and `mHash` members store the comparison and hashing objects, respectively, and `mSize` stores the number of elements currently in the container.

The Constructor

The constructor initializes all the fields and resizes the `mBuckets` vector to the correct size. Unfortunately, the template syntax is somewhat dense. If the syntax confuses you, consult Chapter 11 for details on writing class templates.

```
// Resize mBuckets with the number of buckets.
template <typename Key, typename T, typename Compare, typename Hash>
```

```

hash_map<Key, T, Compare, Hash>::hash_map(
    const Compare& comp, size_t numBuckets, const Hash& hash)
: mSize(0), mComp(comp), mHash(hash)
{
    if (numBuckets == 0) {
        throw std::invalid_argument("Number of buckets must be positive");
    }
    mBuckets.resize(numBuckets);
}

```

The implementation requires at least one bucket, so the constructor enforces that restriction.

Element Lookup

Each of the three major operations (lookup, insertion, and deletion) requires code to find an element with a given key. Thus, it is helpful to have a `private` helper method that performs that task. `findElement()` first uses the `hash` object to calculate the hash of the key and limits the calculated hash value to the number of hash buckets by taking the modulo of the calculated value. Then, it looks in that bucket for an element with a key matching the given key. The elements stored are key/value pairs, so the actual comparison must be done on the first field of the element. The comparison function object specified in the constructor is used to perform the comparison. `lists` require a linear search to find matching elements, but you could use the `find()` algorithm instead of an explicit `for` loop.

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::ListType::iterator
hash_map<Key, T, Compare, Hash>::findElement(const key_type& k, size_t& bucket)
{
    // Hash the key to get the bucket.
    bucket = mHash(k) % mBuckets.size();

    // Look for the key in the bucket.
    for (auto it = std::begin(mBuckets[bucket]);
        it != std::end(mBuckets[bucket]); ++it) {
        if (mComp(it->first, k)) {
            return it;
        }
    }
    return std::end(mBuckets[bucket]);
}

```

Note that `findElement()` returns an iterator referring to an element in the `list` representing the bucket to which the key hashed. If the element is found, the iterator refers to that element; otherwise, it is the end iterator for that `list`. The bucket is returned by reference in the `bucket` argument.

The syntax in this method is somewhat confusing, particularly the use of the `typename` keyword. You must use the `typename` keyword whenever you are using a type that is dependent on a template parameter. Specifically, the type `ListType::iterator`, which is `list<pair<const Key, T>>::iterator`, is dependent on both the `Key` and `T` template parameters.

You can implement the `find()` method as a simple wrapper for `findElement()`:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::value_type*
hash_map<Key, T, Compare, Hash>::find(const key_type& k)
{
    size_t bucket;
    // Use the findElement() helper.
    auto it = findElement(k, bucket);
    if (it == std::end(mBuckets[bucket])) {
        // Element not found -- return nullptr.
        return nullptr;
    }
    // Element found -- return a pointer to it.
    return &(*it);
}
```

The `const` version of `find()` uses a `const_cast` to forward the request to the non-`const` version to avoid code duplication:

```
template <typename Key, typename T, typename Compare, typename Hash>
const typename hash_map<Key, T, Compare, Hash>::value_type*
hash_map<Key, T, Compare, Hash>::find(const key_type& k) const
{
    return const_cast<hash_map<Key, T, Compare, Hash>*>(this)->find(k);
}
```

The `operator[]` method uses the `find()` method, and if it does not find the element, it inserts it as follows:

```
template <typename Key, typename T, typename Compare, typename Hash>
T& hash_map<Key, T, Compare, Hash>::operator[](const key_type& k)
{
    // Try to find the element. If it doesn't exist, add a new element.
    value_type* found = find(k);
    if (found == nullptr) {
        insert(std::make_pair(k, T()));
        found = find(k);
    }
    return found->second;
}
```

This implementation is somewhat inefficient, because in the worst case it calls `find()` twice and `insert()` once. Implementing this more efficiently is a good exercise for the reader.

Element Insert

`insert()` must first check if an element with that key is already in the `hash_map`. If not, it can add the element to the `list` in the appropriate bucket. Note that `findElement()` returns by reference the bucket to which the key hashes, even if the element with that key is not found:

```
template <typename Key, typename T, typename Compare, typename Hash>
void hash_map<Key, T, Compare, Hash>::insert(const value_type& x)
{
```

```

        size_t bucket;
        // Try to find the element.
        auto it = findElement(x.first, bucket);
        if (it != std::end(mBuckets[bucket])) {
            // The element already exists.
            return;
        } else {
            // We didn't find the element, so insert a new one.
            mSize++;
            mBuckets[bucket].push_back(x);
        }
    }
}

```

Element Delete

`erase()` follows the same pattern as `insert()`: It first attempts to find the element by calling `findElement()`. If the element exists, it erases it from the list in the appropriate bucket. Otherwise, it does nothing.

```

template <typename Key, typename T, typename Compare, typename Hash>
void hash_map<Key, T, Compare, Hash>::erase(const key_type& k)
{
    size_t bucket;
    // First, try to find the element.
    auto it = findElement(k, bucket);
    if (it != std::end(mBuckets[bucket])) {
        // The element exists -- erase it.
        mBuckets[bucket].erase(it);
        mSize--;
    }
}

```

Using the Basic Hash Map

Here is a small test program demonstrating the basic `hash_map` class template:

```

hash_map<int, int> myHash;
myHash.insert(make_pair(4, 40));
myHash.insert(make_pair(6, 60));
// x will have type hash_map<int, int>::value_type*
auto x = myHash.find(4);
if (x != nullptr) {
    cout << "4 maps to " << x->second << endl;
} else {
    cout << "cannot find 4 in map" << endl;
}
myHash.erase(4);
auto x2 = myHash.find(4);
if (x2 != nullptr) {
    cout << "4 maps to " << x2->second << endl;
} else {
    cout << "cannot find 4 in map" << endl;
}
myHash[4] = 35;

```

```

myHash[4] = 60;
auto x3 = myHash.find(4);
if (x3 != nullptr) {
    cout << "4 maps to " << x3->second << endl;
} else {
    cout << "cannot find 4 in map" << endl;
}

```

The output is as follows:

```

4 maps to 40
cannot find 4 in map
4 maps to 60

```

Making `hash_map` an STL Container

The basic `hash_map` shown in the previous section follows the spirit, but not the letter, of the STL. For most purposes, the preceding implementation is good enough. However, if you want to use the STL algorithms on your `hash_map`, you must do a bit more work. The C++ standard specifies methods and type aliases that a data structure must provide in order to qualify as an STL container.

Type Alias Container Requirements

The C++ standard specifies that every STL container must provide the following public type aliases:

TYPE NAME	DESCRIPTION
<code>value_type</code>	The element type stored in the container
<code>reference</code>	A reference to the element type stored in the container
<code>const_reference</code>	A reference to a <code>const</code> element type stored in the container
<code>iterator</code>	The type for iterating over elements of the container
<code>const_iterator</code>	A version of <code>iterator</code> for iterating over <code>const</code> elements of the container
<code>size_type</code>	A type that can represent the number of elements in the container; usually just <code>size_t</code> (from <code><cstddef></code>)
<code>difference_type</code>	A type that can represent the difference of two <code>iterators</code> for the container; usually just <code>ptrdiff_t</code> (from <code><cstddef></code>)

Here are the definitions in the `hash_map` class of all these type aliases except `iterator` and `const_iterator`. Writing an iterator is covered in detail in a subsequent section. Note that `value_type` (plus `key_type` and `mapped_type`, which are discussed later) was already defined in the previous version of the `hash_map`. This implementation also adds a type alias `hash_map_type` to give a shorter name to a specific template instantiation of `hash_map`.

```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = hash<Key>>
class hash_map
{
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;
    using reference = std::pair<const Key, T>&;
    using const_reference = const std::pair<const Key, T>&;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using hash_map_type = hash_map<Key, T, Compare, Hash>;
    // Remainder of class definition omitted for brevity
};

```

Method Container Requirements

In addition to the type aliases, every container must provide the following methods:

METHOD	DESCRIPTION	WORST CASE COMPLEXITY
Default Constructor	Constructs an empty container	Constant
Copy constructor	Performs a deep copy of the container	Linear
Move constructor	Performs a move constructing operation	Constant
Copy Assignment operator	Performs a deep copy of the container	Linear
Move Assignment operator	Performs a move assignment operation	Constant
Destructor	Destroys dynamically allocated memory; calls destructor on all elements left in container	Linear
iterator begin(); const_iterator begin() const;	Returns an iterator referring to the first element in the container	Constant
iterator end(); const_iterator end() const;	Returns an iterator referring to one-past-the-last element in the container	Constant
const_iterator cbegin() const;	Returns a const iterator referring to the first element in the container	Constant
const_iterator cend() const;	Returns a const iterator referring to one-past-the-last element in the container	Constant

METHOD	DESCRIPTION	WORST CASE COMPLEXITY
operator== operator!= operator< operator> operator<= operator>=	Comparison operators that compare two containers, element by element	Linear
void swap(Container&);	Swaps the contents of the container passed to the method with the object on which the method is called	Constant
size_type size() const;	Returns the number of elements in the container	Constant
size_type max_size() const;	Returns the maximum number of elements the container can hold	Constant
bool empty() const;	Specifies whether the container has any elements	Constant

NOTE *In this hash_map example, comparison operators are omitted. Implementing them would be a good exercise for the reader.*

The following code extract shows the declarations of all the remaining methods except for `begin()`, `end()`, `cbegin()`, and `cend()`. Those are covered in the next section. The implementation discussed in this text omits the copy and move constructor, and the copy and move assignment operators. They are required if you want to copy, move, or assign `hash_maps`. The downloadable source code contains an implementation of those constructors and operators, in case you are interested.

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
         typename Hash = hash<Key>>
class hash_map
{
public:
    // Type aliases and constructor omitted for brevity

    // Size methods
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    // Other modifying utilities
    void swap(hash_map_type& hashIn);

    // Other methods omitted for brevity
};
```

The implementations of `size()` and `empty()` are easy because the `hash_map` implementation tracks its size in the `mSize` data member. Note that `size_type` is one of the type aliases defined in the class. Because it is a member of the class, such a return type in the implementation must be fully qualified with `typename hash_map<Key, T, Compare, Hash>:`

```
template <typename Key, typename T, typename Compare, typename Hash>
bool hash_map<Key, T, Compare, Hash>::empty() const
{
    return mSize == 0;
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::size_type
hash_map<Key, T, Compare, Hash>::size() const
{
    return mSize;
}
```

`max_size()` is a little trickier. At first, you might think the maximum size of a `hash_map` container is the sum of the maximum size of all the lists. However, the worst-case scenario is that all the elements hash to the same bucket. Thus, the maximum size it can claim to support is the maximum size of a single list:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::size_type
hash_map<Key, T, Compare, Hash>::max_size() const
{
    // In the worst case, all the elements hash to the same
    // list, so the max_size is the max_size of a single list.
    // This code assumes that all the lists have the same max_size.
    return mBuckets[0].max_size();
}
```

Finally, the implementation of `swap()` uses the `std::swap()` utility function to swap each of the four data members.

```
// Just swap the four data members. Use the generic swap template.
template <typename Key, typename T, typename Compare, typename Hash>
void hash_map<Key, T, Compare, Hash>::swap(hash_map_type& hashIn)
{
    // Explicitly qualify with std:: so the compiler doesn't think
    // it's a recursive call.
    std::swap(*this, hashIn);
}
```

Writing an Iterator

The most important container requirement is the iterator. In order to work with the generic algorithms, every container must provide an iterator for accessing the elements in the container. Your iterator should generally provide overloaded `operator*` and `operator->`, plus some other operations depending on its specific behavior. As long as your iterator provides the basic iteration operations, everything should be fine.

The first decision to make about your iterator is what kind it will be: forward, bidirectional, or random access. Random-access iterators don't make much sense for associative containers, so bidirectional seems like the logical choice for a `hash_map` iterator. That means you must also provide `operator++`, `operator--`, `operator==`, and `operator!=`. Consult Chapter 16 for more details on the requirements for the different iterators.

The second decision is how to order the elements of your container. The `hash_map` is unsorted, so iterating in a sorted order is probably too difficult. Instead, your iterator can just step through the buckets, starting with the elements in the first bucket and progressing to those in the last bucket. This order will appear random to the client, but will be consistent and repeatable.

The third decision is how to represent your iterator internally. The implementation is usually quite dependent on the internal implementation of the container. The first purpose of an iterator is to refer to a single element in the container. In the case of a `hash_map`, each element is in an STL list, so perhaps a `hash_map` iterator can be a wrapper around a `list` iterator referring to the element in question. However, the second purpose of a bidirectional iterator is to allow the client to progress to the next or previous element from the current. In order to progress from one bucket to the next, you need to track the current bucket and the `hash_map` object to which the iterator refers.

Once you've chosen your implementation, you must decide on a consistent representation for the end iterator. Recall that the end iterator should really be the "past-the-end" marker: the iterator that's reached by applying `++` to an iterator referring to the final element in the container. A `hash_map` iterator can use as its end iterator the end iterator of the `list` of the final bucket in the `hash_map`.

A container needs to provide both a `const` iterator and a non-`const` iterator. The non-`const` iterator must be convertible to a `const` iterator. This implementation defines a `const_hash_map_iterator` class with `hash_map_iterator` deriving from it.

The `const_hash_map_iterator` Class

Given the decisions made in the previous section, it's time to define the `const_hash_map_iterator` class. The first thing to note is that each `const_hash_map_iterator` object is an iterator for a specific instantiation of the `hash_map` class. In order to provide this one-to-one mapping, the `const_hash_map_iterator` must also be a class template with the hash map type as a template parameter called `HashMap`.

The main question in the class definition is how to conform to the bidirectional iterator requirements. Recall that anything that behaves like an iterator is an iterator. Your class is not required to derive from another class in order to qualify as a bidirectional iterator. However, if you want your iterator to be usable in the generic algorithms functions, you must specify its traits. The discussion earlier in this chapter explains that `iterator_traits` is a class template that defines five `typedefs` for each iterator type. It can be partially specialized for your new iterator type if you want. Alternatively, the default implementation of the `iterator_traits` class template just grabs the five `typedefs` out of the iterator class itself. Thus, you can define those `typedefs` directly in your iterator class. In fact, C++ makes it even easier than that. Instead of defining them yourself, you can just derive from the `iterator` class template, which provides the `typedefs` for you. That way you only need to specify the iterator type and the element type as template arguments to the iterator class template. The `const_hash_map_iterator` is a bidirectional iterator, so you specify `bidirectional_iterator_tag` as the iterator type. Other legal iterator

types are `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, and `random_access_iterator_tag`. For the `const_hash_map_iterator`, the element type is `typename HashMap::value_type`.

Basically, it all boils down to the fact that it's easiest if you derive your iterator classes from the generic iterator class template.

Here is the basic `const_hash_map_iterator` class definition:

```
template <typename HashMap>
class const_hash_map_iterator : public std::iterator<
    std::bidirectional_iterator_tag, typename HashMap::value_type>
{
public:
    using value_type = typename HashMap::value_type;
    using list_iterator_type = typename HashMap::ListType::const_iterator;

    // Bidirectional iterators must supply default ctor
    const_hash_map_iterator();
    const_hash_map_iterator(size_t bucket, list_iterator_type listIt,
                           const HashMap* inHashmap);

    const value_type& operator*() const;

    // Return type must be something to which -> can be applied.
    // Return a pointer to a pair<const Key, T>, to which the compiler will
    // apply -> again.
    const value_type* operator->() const;

    const_hash_map_iterator<HashMap>& operator++();
    const_hash_map_iterator<HashMap> operator++(int);

    const_hash_map_iterator<HashMap>& operator--();
    const_hash_map_iterator<HashMap> operator--(int);

    // Don't need to define a copy constructor or operator= because the
    // default behavior is what we want.

    // Don't need destructor because the default behavior
    // (not deleting mHashmap) is what we want.

    // The following are ok as member functions because we don't
    // support comparisons of different types to this one.
    bool operator==(const const_hash_map_iterator<HashMap>& rhs) const;
    bool operator!=(const const_hash_map_iterator<HashMap>& rhs) const;

protected:
    size_t mBucketIndex;
    list_iterator_type mListIterator;
    const HashMap* mHashmap;

    // Helper methods for operator++ and operator--
    void increment();
    void decrement();
};

};
```

If the definitions and implementations (shown in the next section) of the overloaded operators confuse you, consult Chapter 14 for details on operator overloading.

The `const_hash_map_iterator` Method Implementations

The `const_hash_map_iterator` constructors initialize the three member variables. The default constructor exists only so that clients can declare `const_hash_map_iterator` variables without initializing them. An iterator constructed with the default constructor does not need to refer to any value, and attempting any operations on it is allowed to have undefined results:

```
// Dereferencing or incrementing an iterator constructed with the default
// ctor is undefined, so it doesn't matter what values we give here.
template<typename HashMap>
const_hash_map_iterator<HashMap>::const_hash_map_iterator()
    : mBucketIndex(0), mListIterator(list_iterator_type()), mHashmap(nullptr)
{
}

template<typename HashMap>
const_hash_map_iterator<HashMap>::const_hash_map_iterator(size_t bucket,
    list_iterator_type listIt, const HashMap* inHashmap)
    : mBucketIndex(bucket), mListIterator(listIt), mHashmap(inHashmap)
{
}
```

The implementations of the dereferencing operators are concise, but can be tricky. Chapter 14 explains that `operator*` and `operator->` are asymmetric; `operator*` returns the actual underlying value, which in this case is the element to which the iterator refers, while `operator->` must return something to which the arrow operator can be applied again. Thus, it returns a pointer to the element. The compiler then applies `->` to the pointer, which results in accessing a field of the element:

```
// Return the actual element
template<typename HashMap>
const typename const_hash_map_iterator<HashMap>::value_type&
    const_hash_map_iterator<HashMap>::operator*() const
{
    return *mListIterator;
}

// Return the iterator, so the compiler can apply -> to it to access
// the actual desired field.
template<typename HashMap>
const typename const_hash_map_iterator<HashMap>::value_type*
    const_hash_map_iterator<HashMap>::operator->() const
{
    return &(*mListIterator);
}
```

The increment and decrement operators are implemented as follows. They defer the actual incrementing and decrementing to the `increment()` and `decrement()` helper methods:

```
// Defer the details to the increment() helper.
template<typename HashMap>
```

```

const_hash_map_iterator<HashMap>&
    const_hash_map_iterator<HashMap>::operator++()
{
    increment();
    return *this;
}

// Defer the details to the increment() helper.
template<typename HashMap>
const_hash_map_iterator<HashMap>
    const_hash_map_iterator<HashMap>::operator++(int)
{
    auto oldIt = *this;
    increment();
    return oldIt;
}

// Defer the details to the decrement() helper.
template<typename HashMap>
const_hash_map_iterator<HashMap>&
    const_hash_map_iterator<HashMap>::operator--()
{
    decrement();
    return *this;
}

// Defer the details to the decrement() helper.
template<typename HashMap>
const_hash_map_iterator<HashMap>
    const_hash_map_iterator<HashMap>::operator--(int)
{
    auto oldIt = *this;
    decrement();
    return oldIt;
}

```

Incrementing a `const_hash_map_iterator` tells it to refer to the “next” element in the container. This method first increments the `list` iterator, then checks if it has reached the end of its bucket. If so, it finds the next non-empty bucket in the `hash_map` and sets the `list` iterator equal to the start element in that bucket. Note that it can’t simply move to the next bucket, because there might not be any elements in it. If there are no more non-empty buckets, `mListIterator` is set, by the convention chosen for this example, to the end iterator of the last bucket in the `hash_map`, which is the special “end” position of the `const_hash_map_iterator`. Iterators are not required to be any safer than dumb pointers, so error-checking for things like incrementing an iterator already at the end is not required:

```

// Behavior is undefined if mListIterator already refers to the past-the-end
// element, or is otherwise invalid.
template<typename HashMap>
void const_hash_map_iterator<HashMap>::increment()
{
    // mListIterator is an iterator into a single bucket. Increment it.
    ++mListIterator;
}

```

```
// If we're at the end of the current bucket,
// find the next bucket with elements.
auto& buckets = mHashmap->mBuckets;
if (mListIterator == std::end(buckets[mBucketIndex])) {
    for (size_t i = mBucketIndex + 1; i < buckets.size(); i++) {
        if (!buckets[i].empty()) {
            // We found a non-empty bucket.
            // Make mListIterator refer to the first element in it.
            mListIterator = std::begin(buckets[i]);
            mBucketIndex = i;
            return;
        }
    }
    // No more non-empty buckets. Assign mListIterator to refer to the end
    // iterator of the last list.
    mBucketIndex = buckets.size() - 1;
    mListIterator = std::end(buckets[mBucketIndex]);
}
}
```

Decrement is the inverse of increment: It makes the iterator refer to the “previous” element in the container. However, there is an asymmetry because of the asymmetry between the way the start and end positions are represented: Start is the first element, but end is “one past” the last element. The algorithm for decrement checks first if the underlying list iterator is at the start of its current bucket. If not, it can just be decremented. Otherwise, the code needs to check for the first non-empty bucket before the current one. If one is found, the list iterator must be set to refer to the last element in that bucket, which is the end iterator decremented by one. If no non-empty buckets are found, the decrement is invalid, so the code can do anything it wants (behavior is undefined). Note that the `for` loop needs to use a signed type for its loop variable and not an unsigned type such as `size_t` because the loop uses `--i`:

```
// Behavior is undefined if mListIterator already refers to the first element,
// or is otherwise invalid.
template<typename HashMap>
void const_hash_map_iterator<HashMap>::decrement()
{
    // mListIterator is an iterator into a single bucket.
    // If it's at the beginning of the current bucket, don't decrement it.
    // Instead, try to find a non-empty bucket before the current one.
    auto& buckets = mHashmap->mBuckets;
    if (mListIterator == std::begin(buckets[mBucketIndex])) {
        for (int i = mBucketIndex - 1; i >= 0; --i) {
            if (!buckets[i].empty()) {
                mListIterator = std::end(buckets[i]);
                --mListIterator;
                mBucketIndex = i;
                return;
            }
        }
    }
    // No more non-empty buckets. This is an invalid decrement.
    // Assign mListIterator to refer to the end iterator of the last list.
    mBucketIndex = buckets.size() - 1;
    mListIterator = std::end(buckets[mBucketIndex]);
}
```

```

        } else {
            // We're not at the beginning of the bucket, so just move down.
            --mListIterator;
        }
    }
}

```

Note that both `increment()` and `decrement()` access private members of the `hash_map` class. Thus, the `hash_map` class must declare `const_hash_map_iterator` to be a friend class.

After `increment()` and `decrement()`, `operator==` and `operator!=` are positively simple. They just compare each of the three data members of the objects:

```

template<typename HashMap>
bool const_hash_map_iterator<HashMap>::operator==(const const_hash_map_iterator<HashMap>& rhs) const
{
    // All fields, including the hash_map to which the iterators refer,
    // must be equal.
    return (mHashmap == rhs.mHashmap && mBucketIndex == rhs.mBucketIndex &&
            mListIterator == rhs.mListIterator);
}

template<typename HashMap>
bool const_hash_map_iterator<HashMap>::operator!=(const const_hash_map_iterator<HashMap>& rhs) const
{
    return !(*this == rhs);
}

```

The `hash_map_iterator` Class

The `hash_map_iterator` class derives from `const_hash_map_iterator`. It does not need to override `operator==`, `operator!=`, `increment()`, and `decrement()` because the base class versions are just fine:

```

template <typename HashMap>
class hash_map_iterator : public const_hash_map_iterator<HashMap>
{
public:
    using value_type = typename const_hash_map_iterator<HashMap>::value_type;
    using list_iterator_type = typename HashMap::ListType::iterator;

    // Bidirectional iterators must supply default ctor
    hash_map_iterator();
    hash_map_iterator(size_t bucket, list_iterator_type listIt,
                     HashMap* inHashmap);

    value_type& operator*();

    // Return type must be something to which -> can be applied.
    // Return a pointer to a pair<const Key, T>, to which the compiler will
    // apply -> again.
    value_type* operator->();

    hash_map_iterator<HashMap>& operator++();
}

```

```
    hash_map_iterator<HashMap> operator++(int);

    hash_map_iterator<HashMap>& operator--();
    hash_map_iterator<HashMap> operator--(int);
};
```

The hash_map_iterator Method Implementations

The implementations of the `hash_map_iterator` methods are rather straightforward. The constructors just call the base class constructors. The `operator*` and `operator->` use `const_cast` to return a non-const type. `operator++` and `operator--` just use the `increment()` and `decrement()` from the base class but return a `hash_map_iterator` instead of a `const_hash_map_iterator`. The C++ name lookup rules require you to explicitly use the `this` pointer to refer to data members and methods in a base class template:

```
template<typename HashMap>
hash_map_iterator<HashMap>::hash_map_iterator() : const_hash_map_iterator<HashMap>()
{
}

template<typename HashMap>
hash_map_iterator<HashMap>::hash_map_iterator(size_t bucket,
    list_iterator_type listIt, HashMap* inHashmap)
    : const_hash_map_iterator<HashMap>(bucket, listIt, inHashmap)
{
}

// Return the actual element.
template<typename HashMap>
typename hash_map_iterator<HashMap>::value_type&
    hash_map_iterator<HashMap>::operator*()
{
    return const_cast<value_type&>(*this->mListIterator);
}

// Return the iterator, so the compiler can apply -> to it to access
// the actual desired field.
template<typename HashMap>
typename hash_map_iterator<HashMap>::value_type*
    hash_map_iterator<HashMap>::operator->()
{
    return const_cast<value_type*>(&(*this->mListIterator));
}

// Defer the details to the increment() helper in the base class.
template<typename HashMap>
hash_map_iterator<HashMap>& hash_map_iterator<HashMap>::operator++()
{
    this->increment();
    return *this;
}
```

```

// Defer the details to the increment() helper in the base class.
template<typename HashMap>
hash_map_iterator<HashMap> hash_map_iterator<HashMap>::operator++(int)
{
    auto oldIt = *this;
    this->increment();
    return oldIt;
}

// Defer the details to the decrement() helper in the base class.
template<typename HashMap>
hash_map_iterator<HashMap>& hash_map_iterator<HashMap>::operator--()
{
    this->decrement();
    return *this;
}

// Defer the details to the decrement() helper in the base class.
template<typename HashMap>
hash_map_iterator<HashMap> hash_map_iterator<HashMap>::operator--(int)
{
    auto oldIt = *this;
    this->decrement();
    return oldIt;
}

```

Iterator Type Aliases and Access Methods

The final piece involved in providing iterator support for `hash_map` is to supply the necessary type aliases in the `hash_map` class definition and to write the `begin()`, `end()`, `cbegin()`, and `cend()` methods on the `hash_map`. The type aliases and method prototypes look like this:

```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
         typename Hash = hash<Key>>
class hash_map
{
public:
    // Other type aliases omitted for brevity

    using iterator = hash_map_iterator<hash_map_type>;
    using const_iterator = const_hash_map_iterator<hash_map_type>;

    // Iterator methods
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    const_iterator cbegin() const;
    const_iterator cend() const;

    // Remainder of class definition omitted for brevity
};

```

The trickiest aspect of `begin()` is to remember to return the end iterator if there are no elements in the container:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
    hash_map<Key, T, Compare, Hash>::begin()
{
    if (mSize == 0) {
        // Special case: there are no elements, so return the end iterator.
        return end();
    }

    // We know there is at least one element. Find the first element.
    for (size_t i = 0; i < mBuckets.size(); ++i) {
        if (!mBuckets[i].empty()) {
            return hash_map_iterator<hash_map_type>(i,
                std::begin(mBuckets[i]), this);
        }
    }
    // Should never reach here, but if we do, return the end iterator.
    return end();
}
end() creates a hash_map_iterator referring to the end iterator of the last bucket:
```

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
    hash_map<Key, T, Compare, Hash>::end()
{
    // The end iterator is the end iterator of the list of the last bucket.
    size_t bucket = mBuckets.size() - 1;
    return hash_map_iterator<hash_map_type>(bucket,
        std::end(mBuckets[bucket]), this);
}
```

The implementations of the `const` versions of `begin()` and `end()` use `const_cast` to call the non-`const` versions. These non-`const` versions return a `hash_map_iterator`, which is convertible to a `const_hash_map_iterator`:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::const_iterator
    hash_map<Key, T, Compare, Hash>::begin() const
{
    return const_cast<hash_map_type*>(this)->begin();
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::const_iterator
    hash_map<Key, T, Compare, Hash>::end() const
{
    return const_cast<hash_map_type*>(this)->end();
}
```

The `cbegin()` and `cend()` methods forward the request to the `const` versions of `begin()` and `end()`:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::const_iterator
    hash_map<Key, T, Compare, Hash>::cbegin() const
{
    return begin();
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::const_iterator
    hash_map<Key, T, Compare, Hash>::cend() const
{
    return end();
}
```

Using the `hash_map` Iterators

Now that the `hash_map` supports iteration, you can iterate over its elements just as you would on any STL container, and you can pass the iterators to methods and functions:

```
hash_map<string, int> myHash;
myHash.insert(make_pair("KeyOne", 100));
myHash.insert(make_pair("KeyTwo", 200));
myHash.insert(make_pair("KeyThree", 300));

for (auto it = myHash.cbegin(); it != myHash.cend(); ++it) {
    // Use both -> and * to test the operations.
    cout << it->first << " maps to " << (*it).second << endl;
}

// Print elements using range-based for loop
for (auto& p : myHash) {
    cout << p.first << " maps to " << p.second << endl;
}

// Create a map with all the elements in the hash_map.
map<string, int> myMap(cbegin(myHash), cend(myHash));
for (auto& p : myMap) {
    cout << p.first << " maps to " << p.second << endl;
}
```

This piece of code also shows that the non-member functions such as `std::cbegin()` and `std::cend()` are working as expected.

Note on Allocators

As described earlier in this chapter, all the STL containers allow you to specify a custom memory allocator. A “good citizen” `hash_map` implementation should do the same. However, those details are omitted because they obscure the main points of this implementation.

Note on Reversible Containers

If your container supplies a bidirectional or random access iterator, it is considered *reversible*. Reversible containers are supposed to supply two additional type aliases:

TYPE NAME	DESCRIPTION
reverse_iterator	The type for iterating over elements of the container in reverse order.
const_reverse_iterator	A version of <code>reverse_iterator</code> for iterating over <code>const</code> elements of the container in reverse order.

Additionally, the container should provide `rbegin()` and `rend()`, which are symmetric with `begin()` and `end()`; and should provide `crbegin()` and `crend()`, which are symmetric with `cbegin()` and `cend()`. The usual implementations just use the `reverse_iterator` adapter described earlier in this chapter. These are left as an exercise for the reader.

Making `hash_map` an Associative Container

In addition to the basic container requirements shown already, you can also make your container adhere to additional requirements for associative, unordered associative, or sequential containers. This example makes `hash_map` an associative container, so it should conform to the following type aliases and methods.

Associative Container Type Aliases Requirements

Associative containers require three additional types:

TYPE NAME	DESCRIPTION
<code>key_type</code>	The key type with which the container is instantiated.
<code>key_compare</code>	The comparison class or function pointer type with which the container is instantiated.
<code>value_compare</code>	Class for comparing two <code>value_type</code> elements.

This example implementation also throws in a type alias called `mapped_type`, because that's what `map` does. The `value_compare` is implemented not as a type alias, but as a nested class definition. Alternatively, the class could be a `friend` class of `hash_map`, but this definition follows the `map` definition found in the standard. The purpose of the `value_compare` class is to call the comparison function on the keys of two elements:

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = hash<Key>>
class hash_map
{
public:
    using key_type = Key;
```

```

using mapped_type = T;
using value_type = std::pair<const Key, T>;
using key_compare = Compare;
using reference = std::pair<const Key, T> &;
using const_reference = const std::pair<const Key, T> &;
using size_type = size_t;
using difference_type = ptrdiff_t;
using hash_map_type = hash_map<Key, T, Compare, Hash>;
using iterator = hash_map_iterator<hash_map_type>;
using const_iterator = const_hash_map_iterator<hash_map_type>;

// Required class definition for associative containers
class value_compare {
public:
    public std::binary_function<value_type, value_type, bool>
    {
public:
    bool operator()(const value_type& x, const value_type& y) const
    {
        return comp(x.first, y.first);
    }

protected:
    Compare comp;
    value_compare(Compare c) : comp(c) { }
};

// Remainder of hash_map class definition omitted for brevity
};

```

Associative Container Method Requirements

The standard prescribes quite a few additional method requirements for associative containers:

METHOD	DESCRIPTION	WORST CASE COMPLEXITY
Constructor taking an iterator range.	<p>Constructs the container and inserts elements in the iterator range. The iterator range need not refer to another container of the same type.</p> <p>Note that all constructors of associative containers must take a comparison object. The constructors should provide a default constructed object as the default value.</p>	$n \log n$
Constructor taking an <code>initializer_list<value_type></code> as parameter.	Constructs the container and inserts the elements from the initializer list into the container.	$n \log n$

METHOD	DESCRIPTION	WORST CASE COMPLEXITY
Assignment operator with an <code>initializer_list<value_type></code> as right-hand side.	Replaces all elements from the container with the elements from the initializer list.	$n \log n$
<code>key_compare key_comp () const;</code> <code>value_compare value_comp () const;</code>	Returns the comparison objects for comparing just keys or entire values.	constant
<code>pair<iterator, bool> insert (value_type&);</code> <code>iterator insert (iterator position, value_type&);</code> <code>void insert (InputIterator start, InputIterator end);</code>	Three different forms of <code>insert</code> . The position in the second is a hint, which can be ignored. The range in the third need not be from a container of the same type. Containers that allow duplicate keys return just <code>iterator</code> from the first form, because <code>insert ()</code> always succeeds.	logarithmic, except for the second form, which can be amortized constant if the element is inserted immediately before the given hint.
<code>void insert (initializer_list <value_type>);</code>	Inserts the elements from the initializer list into the container.	logarithmic
<code>pair<iterator, bool> emplace (value_type&&);</code> <code>iterator emplace_hint (iterator hint, value_type&&);</code>	Implements the <code>emplace</code> operations to construct objects <i>in-place</i> . <i>In-place</i> construction is discussed in Chapter 16.	logarithmic, except for the second form, which can be amortized constant if the element is inserted immediately before the given hint.
<code>size_type erase (key_type&);</code> <code>iterator erase (iterator position);</code> <code>iterator erase (iterator start, iterator end);</code>	Three different forms of <code>erase</code> . The first form returns the number of values erased (0 or 1 in containers that do not allow duplicate keys). The second and third forms erase the elements at <code>position</code> , or in the range <code>start</code> to <code>end</code> , and return an iterator to the element following the last erased element.	logarithmic, except for the second form, which should be amortized constant
<code>void clear();</code>	Erases all elements.	linear

METHOD	DESCRIPTION	WORST CASE COMPLEXITY
<pre>iterator find(key_type&); const_iterator find(key_type&) const;</pre>	Finds the element with the specified key.	logarithmic
<pre>size_type count(key_type&) const;</pre>	Returns the number of elements with the specified key (0 or 1 in containers that do not allow duplicate keys).	logarithmic
<pre>pair<iterator,iterator> equal_range(key_type&); pair<const_iterator, const_iterator> equal_range(key_type&) const;</pre>	Returns iterators referring to the first element with the specified key and one past the last element with the specified key.	logarithmic

Note that `hash_map` does not allow duplicate keys, thus `equal_range()` always returns a pair of identical iterators.

Here is the complete `hash_map` class definition. The prototypes for `insert()`, `erase()`, and `find()` need to change slightly from the previous versions shown because those initial versions don't have the right return types required for associative containers:

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = hash<Key>>
class hash_map
{
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;
    using key_compare = Compare;
    using reference = std::pair<const Key, T>&;
    using const_reference = const std::pair<const Key, T>&;
    using size_type = size_t;
    using difference_type = ptrdiff_t;
    using hash_map_type = hash_map<Key, T, Compare, Hash>;
    using iterator = hash_map_iterator<hash_map_type>;
    using const_iterator = const_hash_map_iterator<hash_map_type>;
```

```
// Required class definition for associative containers
class value_compare :
    public std::binary_function<value_type, value_type, bool>
{
public:
    bool operator()(const value_type& x, const value_type& y) const
    {
        return comp(x.first, y.first);
    }

protected:
    Compare comp;
    value_compare(Compare c) : comp(c) { }
};

// The iterator classes need access to all members of the hash_map
friend class hash_map_iterator<hash_map_type>;
friend class const_hash_map_iterator<hash_map_type>;

virtual ~hash_map(); // Virtual destructor

// Throws invalid_argument if the number of buckets is illegal.
explicit hash_map(const Compare& comp = Compare(),
    size_type numBuckets = 101, const Hash& hash = Hash());

// Throws invalid_argument if the number of buckets is illegal.
template <typename InputIterator>
hash_map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    size_type numBuckets = 101, const Hash& hash = Hash());

// Initializer list constructor
// Throws invalid_argument if the number of buckets is illegal.
explicit hash_map(std::initializer_list<value_type> il,
    const Compare& comp = Compare(), size_type numBuckets = 101,
    const Hash& hash = Hash());

// Initializer list assignment operator
hash_map_type& operator=(std::initializer_list<value_type> il);

// Iterator methods
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// Size methods
bool empty() const;
size_type size() const;
size_type max_size() const;

// Element insert methods
T& operator[](const key_type& x);
std::pair<iterator, bool> insert(const value_type& x);
```

```

iterator insert(iterator position, const value_type& x);
template <typename InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(std::initializer_list<value_type> il);

// Emplace methods
std::pair<iterator, bool> emplace(value_type&& x);
iterator emplace_hint(iterator hint, value_type&& x);

// Element delete methods
size_type erase(const key_type& x);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);

// Other modifying utilities
void swap(hash_map_type& hashIn);
void clear() noexcept;

// Access methods for STL conformity
key_compare key_comp() const;
value_compare value_comp() const;

// Lookup methods
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
std::pair<iterator, iterator> equal_range(const key_type& x);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& x) const;
size_type count(const key_type& x) const;

private:
    using ListType = std::list<value_type>;
    typename ListType::iterator findElement(const key_type& x,
        size_type& bucket);

    std::vector<ListType> mBuckets;
    size_type mSize;
    Compare mComp;
    Hash mHash;
};

};


```

hash_map Constructors

The implementation of the default constructor is shown earlier. The second constructor is a method template so that it can take an iterator range from any container, not just other `hash_map`s. If it were not a method template, it would need to specify the `InputIterator` type explicitly as `hash_map_iterator`, limiting it to iterators from `hash_map`s. Despite the syntax, the implementation is uncomplicated: It initializes all the data members, then calls `insert()` to actually insert all the elements in the specified range:

```

// Make a call to insert() to actually insert the elements.
template <typename Key, typename T, typename Compare, typename Hash>
template <typename InputIterator>
hash_map<Key, T, Compare, Hash>::hash_map(

```

```
InputIterator first, InputIterator last, const Compare& comp,
size_type numBuckets, const Hash& hash)
: mSize(0), mComp(comp), mHash(hash)
{
    if (numBuckets == 0) {
        throw std::invalid_argument("Number of buckets must be positive");
    }
    mBuckets.resize(numBuckets);
    insert(first, last);
}
```

hash_map Initializer List Constructor

Initializer lists are discussed in Chapter 10. Following is the implementation of the `hash_map` constructor that takes an initializer list, which is very similar to the implementation of the constructor accepting an iterator range:

```
template <typename Key, typename T, typename Compare, typename Hash>
hash_map<Key, T, Compare, Hash>::hash_map(std::initializer_list<value_type> il,
    const Compare& comp, size_type numBuckets, const Hash& hash)
: mSize(0), mComp(comp), mHash(hash)
{
    if (numBuckets == 0) {
        throw std::invalid_argument("Number of buckets must be positive");
    }
    mBuckets.resize(numBuckets);
    insert(std::begin(il), std::end(il));
}
```

With this initializer list constructor, a `hash_map` can be constructed as follows:

```
hash_map<string, int> myHash {
    {"KeyOne", 100},
    {"KeyTwo", 200},
    {"KeyThree", 300} };
```

This is much nicer than using `insert()` and `make_pair()` calls:

```
myHash.insert(make_pair("KeyOne", 100));
myHash.insert(make_pair("KeyTwo", 200));
myHash.insert(make_pair("KeyThree", 300));
```

hash_map Initializer List Assignment Operator

Assignment operators can also accept an initializer list on the right-hand side. Following is an implementation of an initializer list assignment operator for `hash_map`. It deletes all the elements from the `hash_map` and then delegates the work to the `insert()` method accepting an iterator range:

```
template <typename Key, typename T, typename Compare, typename Hash>
hash_map<Key, T, Compare, Hash>& hash_map<Key, T, Compare, Hash>::operator=(
    std::initializer_list<value_type> il)
{
```

```

        clear();
        insert(std::begin(il), std::end(il));
        return *this;
    }
}

```

With this assignment operator, you can write code as follows:

```

myHash = {
    { "KeyOne", 100 },
    { "KeyTwo", 200 },
    { "KeyThree", 300 } };

```

hash_map Insertion Operations

In the basic `hash_map` section earlier in this chapter, a simple `insert()` method is given. In this version, four `insert()` methods are provided with additional features:

- The simple `insert()` operation returns a `pair<iterator, bool>`, which indicates both where the item is inserted and whether or not it was newly created.
- The version of `insert()` that takes a position is useless for a `hash_map`, but it is provided for symmetry with other kinds of collections. The position is ignored, and it merely calls the first version.
- The third form of `insert()` is a method template, so elements from arbitrary containers can be inserted into the `hash_map`.
- The last form of `insert()` accepts an `initializer_list<value_type>`.

The first two `insert()` methods are implemented as follows:

```

template <typename Key, typename T, typename Compare, typename Hash>
std::pair<typename hash_map<Key, T, Compare, Hash>::iterator, bool>
hash_map<Key, T, Compare, Hash>::insert(const value_type& x)
{
    size_t bucket;
    // Try to find the element.
    auto it = findElement(x.first, bucket);
    bool inserted = false;
    if (it == std::end(mBuckets[bucket])) {
        // We didn't find the element, so insert a new one.
        it = mBuckets[bucket].insert(std::end(mBuckets[bucket]), x);
        inserted = true;
        mSize++;
    }
    return std::make_pair(
        hash_map_iterator<hash_map_type>(bucket, it, this), inserted);
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
hash_map<Key, T, Compare, Hash>::insert(
    iterator /*position*/, const value_type& x)
{

```

```
    // Completely ignore position.
    return insert(x).first;
}
```

The third form of `insert()` is a method template for the same reason as the constructor shown earlier: It should be able to insert elements by using iterators from containers of any type. The actual implementation uses an `insert_iterator`, described earlier in this chapter:

```
template <typename Key, typename T, typename Compare, typename Hash>
template <typename InputIterator>
void hash_map<Key, T, Compare, Hash>::insert(
    InputIterator first, InputIterator last)
{
    // Copy each element in the range by using an insert_iterator adapter.
    // Give begin() as a dummy position -- insert ignores it anyway.
    std::insert_iterator<hash_map_type> inserter(*this, begin());
    std::copy(first, last, inserter);
}
```

The last insert operation accepts an initializer list. The implementation for `hash_map` simply forwards the work to the `insert()` method accepting an iterator range:

```
template <typename Key, typename T, typename Compare, typename Hash>
void hash_map<Key, T, Compare, Hash>::insert(
    std::initializer_list<value_type> il)
{
    insert(std::begin(il), std::end(il));
}
```

With this `insert()` method, you can write code as follows:

```
myHash.insert({
    { "KeyFour", 400 },
    { "KeyFive", 500 } });
```

hash_map Emplace Operations

Emplace operations construct objects in place. They are discussed in Chapter 16. Their implementation is straightforward in this case because the `emplace` operation can be forwarded to the correct bucket list. The `emplace()` method is implemented exactly the same as the corresponding `insert()` method, except that it calls `emplace()` on the list instead of `insert()`:

```
template <typename Key, typename T, typename Compare, typename Hash>
std::pair<typename hash_map<Key, T, Compare, Hash>::iterator, bool>
hash_map<Key, T, Compare, Hash>::emplace(value_type&& x)
{
    size_t bucket;
    // Try to find the element.
    auto it = findElement(x.first, bucket);
    bool inserted = false;
    if (it == std::end(mBuckets[bucket])) {
        // We didn't find the element, so emplace a new one.
        it = mBuckets[bucket].emplace(std::end(mBuckets[bucket]), x);
        inserted = true;
    }
}
```

```

        mSize++;
    }
    return std::make_pair(
        hash_map_iterator<hash_map_type>(bucket, it, this), inserted);
}

```

The `emplace_hint()` method ignores the hint and forwards the call to the `emplace()` method. This implementation uses `std::forward` to *perfectly forward* `x` to `emplace()`. If an rvalue reference is passed to `emplace_hint()`, then it is forwarded as an rvalue reference to `emplace()`. If an lvalue reference is passed to `emplace_hint()`, then it is forwarded as an lvalue reference:

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
hash_map<Key, T, Compare, Hash>::emplace_hint(
    iterator /*hint*/, value_type&& x)
{
    // Completely ignore hint.
    return emplace(std::forward<value_type>(x)).first;
}

```

hash_map Erase Operations

The version of `erase()` in the earlier section “A Basic Hash Map” is not compliant with STL requirements. You need to implement the following versions:

- A version that takes as a parameter a `key_type` and returns a `size_type` for the number of elements removed from the collection (for `hash_map` there are only two possible return values, 0 and 1)
- A version that erases a value at a specific iterator position, and returns an iterator to the element following the erased element
- A version that erases a range of elements based on two iterators, and returns an iterator to the element following the last erased element

The first version is implemented as follows:

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::size_type
hash_map<Key, T, Compare, Hash>::erase(const key_type& x)
{
    size_t bucket;
    // First, try to find the element.
    auto it = findElement(x, bucket);
    if (it != std::end(mBuckets[bucket])) {
        // The element exists -- erase it.
        mBuckets[bucket].erase(it);
        mSize--;
        return 1;
    } else {
        return 0;
    }
}

```

The second form of `erase()` must remove the element at a specific iterator position. The iterator given is, of course, a `hash_map_iterator`. Thus, `hash_map` must have some ability to obtain the underlying bucket and `list` iterator from the `hash_map_iterator`. The approach taken is to make the `hash_map` class a friend of the `hash_map_iterator` (not shown in the earlier class definition):

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
    hash_map<Key, T, Compare, Hash>::erase(iterator position)
{
    iterator next = position;
    ++next;
    // Erase the element from its bucket.
    mBuckets[position.mBucketIndex].erase(position.mListIterator);
    mSize--;
    return next;
}
```

The final version of `erase()` removes a range of elements. It iterates from `first` to `last`, calling `erase()` on each element, thus letting the previous version of `erase()` do all the work:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
    hash_map<Key, T, Compare, Hash>::erase(iterator first, iterator last)
{
    // Erase all the elements in the range.
    for (iterator next = first; next != last;) {
        next = erase(next);
    }
    return last;
}
```

hash_map Clear Operation

The `clear()` method uses a range-based `for` loop to call `clear()` on the `list` representing each bucket:

```
template <typename Key, typename T, typename Compare, typename Hash>
void hash_map<Key, T, Compare, Hash>::clear() noexcept
{
    // Call clear on each bucket.
    for (auto& bucket : mBuckets) {
        bucket.clear();
    }
    mSize = 0;
}
```

hash_map Accessor Operations

The standard requires access methods for the key comparison and value comparison objects. These methods must be called `key_comp()` and `value_comp()`:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::key_compare
    hash_map<Key, T, Compare, Hash>::key_comp() const
```

```

    {
        return mComp;
    }

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::value_compare
    hash_map<Key, T, Compare, Hash>::value_compare() const
{
    return value_compare(mComp);
}

```

The `find()` method is identical to the version shown earlier for the basic `hash_map`, except for the return code. Instead of returning a pointer to the element, it constructs a `hash_map_iterator` referring to it:

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::iterator
    hash_map<Key, T, Compare, Hash>::find(const key_type& x)
{
    size_t bucket;
    // Use the findElement() helper.
    auto it = findElement(x, bucket);
    if (it == std::end(mBuckets[bucket])) {
        // Element not found -- return the end iterator.
        return end();
    }
    // Element found -- convert the bucket/iterator to a hash_map_iterator.
    return hash_map_iterator<hash_map_type>(bucket, it, this);
}

```

The `const` version of `find()` is identical except that it returns a `const_hash_map_iterator`, and uses `const_cast` to avoid having to duplicate the `findElement()` helper method:

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::const_iterator
    hash_map<Key, T, Compare, Hash>::find(const key_type& x) const
{
    size_t bucket;
    // Use the findElement() helper.
    auto it = const_cast<hash_map_type*>(this)->findElement(x, bucket);
    if (it == std::end(mBuckets[bucket])) {
        // Element not found -- return the end iterator.
        return end();
    }
    // Element found -- convert bucket/iterator to a const_hash_map_iterator.
    return const_hash_map_iterator<hash_map_type>(bucket, it, this);
}

```

The implementations of both versions of `equal_range()` are identical except one returns a pair of `hash_map_iterators` while the other returns a pair of `const_hash_map_iterators`. They both simply forward the request to `find()`. A `hash_map` cannot have elements with duplicate keys, so the result of `equal_range()` for `hash_map` is always a pair of identical iterators:

```
template <typename Key, typename T, typename Compare, typename Hash>
```

```
std::pair<
    typename hash_map<Key, T, Compare, Hash>::iterator,
    typename hash_map<Key, T, Compare, Hash>::iterator>
hash_map<Key, T, Compare, Hash>::equal_range(const key_type& x)
{
    auto it = find(x);
    return std::make_pair(it, it);
}
```

The implementation of `count()` is a wrapper for `find()`, returning 1 if it finds the element, and 0 if it doesn't. The `find()` method returns the end iterator if it can't find the element. `count()` retrieves an end iterator by calling `end()` in order to compare it. Since this `hash_map` does not allow duplicate keys, `count()` always returns either 0 or 1:

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hash_map<Key, T, Compare, Hash>::size_type
hash_map<Key, T, Compare, Hash>::count(const key_type& x) const
{
    // There are either 1 or 0 elements matching key x.
    // If we can find a match, return 1, otherwise return 0.
    if (find(x) == end()) {
        return 0;
    } else {
        return 1;
    }
}
```

The final method, `operator[]`, is not required by the standard, but is provided for convenience of the programmer, and to be symmetric with `std::map`. The prototype and implementation are identical to those of the `operator[]` in `std::map`. The comments explain the potentially confusing one-line implementation:

```
template <typename Key, typename T, typename Compare, typename Hash>
T& hash_map<Key, T, Compare, Hash>::operator[](const key_type& x)
{
    // This definition is the same as that used by map, according to
    // the standard.
    // It's a bit cryptic, but it basically attempts to insert
    // a new key/value pair of x and a new value. Regardless of whether
    // the insert succeeds or fails, insert() returns a pair of an
    // iterator/bool. The iterator refers to a key/value pair, the
    // second element of which is the value we want to return.
    return ((insert(std::make_pair(x, T()))).first)->second;
}
```

Note on Sequential Containers

The `hash_map` developed in the preceding sections is an *associative container*. However, you could also write a *sequential container*, or an *unordered associative container*, in which case you would need to follow a different set of requirements. Instead of listing them here, it's easier to point out that the `deque` container follows the prescribed sequential container requirements almost exactly.

The only difference is that it provides an extra `resize()` method (not required by the standard). An example of an unordered associative container is the `unordered_map`, on which you can model your own unordered associative containers.

SUMMARY

The final example in this chapter showed almost the complete development of a `hash_map` associative container and its iterator. This `hash_map` implementation is given here to teach you how to write your own STL containers and iterators. C++11 includes its own set of unordered associative containers or hash tables. You should use them instead of your own implementation.

In the process of reading this chapter, you hopefully gained an appreciation for the steps involved in developing containers. Even if you never write another STL algorithm or container, you understand better the STL's mentality and capabilities, and you can put it to better use.

This chapter concludes the tour of the STL. Even with all the details given in this book, there are still features omitted. If this material excited you, and you would like more information, consult some of the resources in Appendix B. Don't feel compelled to use all the features discussed here. Forcing them into your programs without a true need will just complicate your code. However, I encourage you to consider incorporating aspects of the STL into your programs where they make sense. Start with the containers, maybe throw in an algorithm or two, and before you know it, you'll be a convert!

21

Advanced Templates

WHAT'S IN THIS CHAPTER?

- The different kinds of template parameters
- How to use partial specialization
- How to write recursive templates
- Explaining variadic templates
- How to write type-safe variable argument functions using variadic templates
- What metaprogramming is and how to use it

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Chapter 11 covered the most widely used features of class and function templates. If you are interested in only a basic knowledge of templates so that you can better understand how the STL works, or perhaps write your own simple classes, you can skip this chapter on advanced templates. However, if templates interest you and you want to uncover their full power, continue reading this chapter to learn about some of the more obscure, but fascinating, details.

MORE ABOUT TEMPLATE PARAMETERS

There are actually three kinds of template parameters: type, non-type, and template template (no, you're not seeing double: that really is the name). You've seen examples of type and non-type parameters in Chapter 11, but not template template parameters yet. There are also some tricky aspects to both type and non-type parameters that are not covered in Chapter 11.

More about Template Type Parameters

Template type parameters are the main purpose of templates. You can declare as many type parameters as you want. For example, you could add to the grid template from Chapter 11 a second type parameter specifying another templatized class container on which to build the grid. The standard template library defines several templatized container classes, including `vector` and `deque`. The original grid class uses a `vector` of `vectors` to store the elements of a grid. A user of the `Grid` class might want to use a `vector` of `deques` instead. With another template type parameter, you can allow the user to specify whether she wants the underlying container to be a `vector` or a `deque`. Here is the class definition with the additional template parameter:

```
template <typename T, typename Container>
class Grid
{
public:
    explicit Grid(size_t inWidth = kDefaultWidth,
                  size_t inHeight = kDefaultHeight);
    virtual ~Grid();

    // Sets an element at a given location. The element is copied.
    void setElementAt(size_t x, size_t y, const T& inElem);
    T& getElementAt(size_t x, size_t y);
    const T& getElementAt(size_t x, size_t y) const;

    size_t getHeight() const { return mHeight; }
    size_t getWidth() const { return mWidth; }
    static const size_t kDefaultWidth = 10;
    static const size_t kDefaultHeight = 10;

private:
    void initializeCellsContainer();
    std::vector<Container> mCells;
    size_t mWidth, mHeight;
};
```

This template now has two parameters: `T` and `Container`. Thus, wherever you previously referred to `Grid<T>` you must now refer to `Grid<T, Container>` to specify both template parameters. The only other change is that `mCells` is now a `vector` of `Containers` instead of a `vector` of `vectors`.

Here is the constructor definition:

```
template <typename T, typename Container>
Grid<T, Container>::Grid(size_t inWidth, size_t inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    initializeCellsContainer();
}
```

The constructor delegates the work to `initializeCellsContainer()`, which is implemented as follows. It assumes that the `Container` type has a `resize()` method. If you try to instantiate this template by specifying a type that has no `resize()` method, the compiler will generate an error.

```

template <typename T, typename Container>
void Grid<T, Container>::initializeCellsContainer()
{
    mCells.resize(mWidth);
    for (auto& column : mCells) {
        column.resize(mHeight);
    }
}

```

Here are the implementations of the remaining methods:

```

template <typename T, typename Container>
void Grid<T, Container>::setElementAt(size_t x, size_t y, const T& inElem)
{
    mCells[x][y] = inElem;
}

template <typename T, typename Container>
T& Grid<T, Container>::getElementAt(size_t x, size_t y)
{
    return mCells[x][y];
}

template <typename T, typename Container>
const T& Grid<T, Container>::getElementAt(size_t x, size_t y) const
{
    return mCells[x][y];
}

```

Now you can instantiate and use `Grid` objects like this:

```

Grid<int, vector<int>> myIntVectorGrid;
Grid<int, deque<int>> myIntDequeGrid;

myIntVectorGrid.setElementAt(3, 4, 5);
cout << myIntVectorGrid.getElementAt(3, 4) << endl;

myIntDequeGrid.setElementAt(1, 2, 3);
cout << myIntDequeGrid.getElementAt(1, 2) << endl;

Grid<int, vector<int>> grid2(myIntVectorGrid);
grid2 = myIntVectorGrid;

```

The use of the word `Container` for the parameter name doesn't mean that the type really must be a container. You could try to instantiate the `Grid` class with an `int` instead:

```
Grid<int, int> test; // WILL NOT COMPILE
```

This line will not compile, but it might not give you the error you expect. It won't complain that the second type argument is an `int` instead of a container. It will tell you something more cryptic. For example, Microsoft Visual C++ tells you that `left of '.resize'` must have `class/struct/union`

type. That's because the compiler attempts to generate a `Grid` class with `int` as the `Container`. Everything works fine until it tries to compile this line:

```
column.resize(mHeight);
```

At that point, the compiler realizes that `column` is an `int`, so you can't call `resize()` on it.

This approach is used in the STL. The `stack`, `queue`, and `priority_queue` class templates all take a template type parameter specifying the underlying container.

Default Values for Template Type Parameters

You can give template parameters default values. For example, you might want to say that the default container for your `Grid` is a `vector`. The class template definition would look like this:

```
template <typename T, typename Container = std::vector<T>>
class Grid
{
    // Everything else is the same as before.
};
```

You can use the type `T` from the first template parameter as the argument to the `vector` template in the default value for the second template parameter. The C++ syntax requires that you do not repeat the default value in the template header line for method definitions. With this default parameter, clients can now instantiate a grid with or without specifying an underlying container:

```
Grid<int, deque<int>> myDequeGrid;
Grid<int, vector<int>> myVectorGrid;
Grid<int> myVectorGrid2;
```

Introducing Template Template Parameters

There is one problem with the `Container` parameter in the previous section. When you instantiate the class template, you write something like this:

```
Grid<int, vector<int>> myIntGrid;
```

Note the repetition of the `int` type. You must specify that it's the element type both of the `Grid` and of the `vector`. What if you wrote this instead?

```
Grid<int, vector<SpreadsheetCell>> myIntGrid;
```

That wouldn't work very well. It would be nice to be able to write the following, so that you couldn't make that mistake:

```
Grid<int, vector> myIntGrid;
```

The `Grid` class should be able to figure out that it wants a `vector` of `ints`. The compiler won't allow you to pass that argument to a normal type parameter though, because `vector` by itself is not a type but a template.

If you want to take a template as a template parameter, you must use a special kind of parameter called a *template template parameter*. Specifying a template template parameter is sort of like

specifying a function pointer parameter in a normal function. Function pointer types include the return type and parameter types of a function. Similarly, when you specify a template template parameter, the full specification of the template template parameter includes the parameters to that template.

For example, containers such as `vector` and `deque` have a template parameter list that looks something as follows. The `E` parameter is the element type. The `Allocator` parameter is covered in Chapter 16.

```
template <typename E, typename Allocator = allocator<E> >
class vector
{
    // Vector definition
};
```

To pass such a container as a template template parameter, all you have to do is copy-paste the declaration of the class template (in this example: `template <typename E, typename Allocator = allocator<E>> class vector`), replace the class name (`vector`) with a parameter name (`Container`), and use that as the template template parameter of another template declaration, `Grid` in this example, instead of a simple type name. Given the preceding template specification, here is the class template definition for the `Grid` class that takes a container template as its second template parameter:

```
template <typename T,
         template <typename E, typename Allocator = std::allocator<E>> class Container
         = std::vector>
class Grid
{
public:
    // Omitted code that is the same as before
private:
    std::vector<Container<T>> mCells;
    // Omitted code that is the same as before
};
```

What is going on here? The first template parameter is the same as before: the element type `T`. The second template parameter is now a template itself for a container such as `vector` or `deque`. As you saw earlier, this “template type” must take two parameters: an element type `E` and an allocator type. Note the repetition of the word `class` after the nested template parameter list. The name of this parameter in the `Grid` template is `Container` (as before). The default value is now `vector`, instead of `vector<T>`, because the `Container` is a template instead of an actual type.

The syntax rule for a template template parameter more generically is this:

```
template <..., template <TemplateTypeParams> class ParameterName, ...>
```

Instead of using `Container` by itself in the code, you must specify `Container<T>` as the container type. For example, the declaration of `mCells` now is as follows:

```
std::vector<Container<T>> mCells;
```

After implementing all the methods, you can use the template as follows:

```
Grid<int, vector> myGrid;
myGrid.setElementAt(1, 2, 3);
cout << myGrid.getElementAt(1, 2) << endl;
Grid<int, vector> myGrid2(myGrid);
```

This C++ syntax is a bit convoluted because it is trying to allow for maximum flexibility. Try not to get bogged down in the syntax here, and keep the main concept in mind: You can pass templates as parameters to other templates.

More about Non-Type Template Parameters

You might want to allow the user to specify a default element used to initialize each cell in the grid. Here is a perfectly reasonable approach to implement this goal. It uses `T()` as the default value for the second template parameter.

```
template <typename T, const T DEFAULT = T()>
class Grid
{
    // Identical as before.
};
```

This definition is legal. You can use the type `T` from the first parameter as the type for the second parameter, and non-type parameters can be `const` just like function parameters. You can use this initial value for `T` to initialize each cell in the grid:

```
template <typename T, const T DEFAULT>
void Grid<T, DEFAULT>::initializeCellsContainer()
{
    mCells.resize(mWidth);
    for (auto& column : mCells) {
        column.resize(mHeight);
        for (auto& element : column) {
            element = DEFAULT;
        }
    }
}
```

The other method definitions stay the same, except that you must add the second template parameter to the template lines, and all the instances of `Grid<T>` become `Grid<T, DEFAULT>`. After making those changes, you can then instantiate an `int` `Grid` with an initial value for all the elements:

```
Grid<int> myIntGrid;           // Initial value is 0
Grid<int, 10> myIntGrid2;     // Initial value is 10
```

The initial value can be any integer you want. However, suppose that you try to create a `SpreadsheetCell` `Grid`:

```
SpreadsheetCell defaultCell;
Grid<SpreadsheetCell, defaultCell> mySpreadsheet; // WILL NOT COMPILE
```

That line leads to a compiler error because you cannot pass objects as arguments to non-type parameters.

WARNING *Non-type parameters cannot be objects, or even doubles or floats. They are restricted to integral types, enums, pointers, and references.*

This example illustrates one of the vagaries of class templates: They can work correctly on one type but fail to compile for another type.

Reference and Pointer Non-Type Template Parameters

A more comprehensive way of allowing the user to specify an initial element value for the grid uses a reference to a `T` as the non-type template parameter. Here is the new class definition:

```
template <typename T, const T& DEFAULT>
class Grid
{
    // Everything else is the same as the previous example.
};
```

Now you can instantiate this class template for any type. However, the C++ standard says that the reference you pass as the second template argument must be a constant expression, and must be a reference to a complete object with static storage duration and external or internal linkage. Chapter 10 discusses external and internal linkage.

Here is a program that declares `int` and `SpreadsheetCell` grids with initial values.

```
namespace {
    const int defaultInt = 11;
    const SpreadsheetCell defaultCell(1.2);
}

int main()
{
    Grid<int, defaultInt> myIntGrid;
    Grid<SpreadsheetCell, defaultCell> mySpreadsheet;
    Grid<int, defaultInt> myIntGrid2(myIntGrid);
    return 0;
}
```

CLASS TEMPLATE PARTIAL SPECIALIZATION

The `const char*` class specialization shown in Chapter 11 is called *full class template specialization* because it specializes the `Grid` template for every template parameter. There are no template parameters left in the specialization. That's not the only way you can specialize a class; you can also write a *partial class specialization*, in which you specialize some template parameters but not

others. For example, recall the basic version of the `Grid` template with `width` and `height` non-type parameters:

```
template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid
{
public:
    Grid();
    virtual ~Grid();

    void setElementAt(size_t x, size_t y, const T& inElem);
    T& getElementAt(size_t x, size_t y);
    const T& getElementAt(size_t x, size_t y) const;

    size_t getHeight() const { return HEIGHT; }
    size_t getWidth() const { return WIDTH; }

private:
    T mCells[WIDTH] [HEIGHT];
};
```

You could specialize this class template for `const char*` C-style strings like this:

```
#include "Grid.h" // The file containing the Grid template definition
template <size_t WIDTH, size_t HEIGHT>
class Grid<const char*, WIDTH, HEIGHT>
{
public:
    Grid();
    virtual ~Grid();

    void setElementAt(size_t x, size_t y, const char* inElem);
    const char* getElementAt(size_t x, size_t y) const;

    size_t getHeight() const { return HEIGHT; }
    size_t getWidth() const { return WIDTH; }

private:
    std::string mCells[WIDTH] [HEIGHT];
};
```

In this case, you are not specializing all the template parameters. Therefore, your template line looks like this:

```
template <size_t WIDTH, size_t HEIGHT>
class Grid<const char*, WIDTH, HEIGHT>
```

Note that the template has only two parameters: `WIDTH` and `HEIGHT`; however, you're writing a `Grid` class for three arguments: `T`, `WIDTH`, and `HEIGHT`. Thus, your template parameter list contains two parameters, and the explicit `Grid<const char*, WIDTH, HEIGHT>` contains three arguments. When

you instantiate the template, you must still specify three parameters. You can't instantiate the template with only height and width:

```
Grid<int, 2, 2> myIntGrid;           // Uses the original Grid
Grid<const char*, 2, 2> myStringGrid; // Uses the partial specialization
Grid<2, 3> test;                   // DOES NOT COMPILE! No type specified.
```

Yes, the syntax is confusing. And it gets worse. In partial specializations, unlike in full specializations, you include the template line in front of every method definition, as in the following example:

```
template <size_t WIDTH, size_t HEIGHT>
const char* Grid<const char*, WIDTH, HEIGHT>::getElementAt(
    size_t x, size_t y) const
{
    return mCells[x][y].c_str();
}
```

You need this template line with two parameters to show that this method is parameterized on those two parameters. Note that wherever you refer to the full class name, you must use `Grid<const char*, WIDTH, HEIGHT>`.

Another Form of Partial Specialization

The previous example does not show the true power of partial specialization. You can write specialized implementations for a subset of possible types without specializing individual types. For example, you can write a specialization of the `Grid` class for all pointer types. The copy constructor and assignment operator of this specialization could perform deep copies of objects to which pointers point instead of shallow copies.

Here is the class definition, assuming that you're specializing the initial version of `Grid` with only one parameter. In this implementation, `Grid` becomes the owner of supplied pointers, so it automatically frees the memory when necessary:

```
#include "Grid.h"
#include <memory>
template <typename T>
class Grid<T*>
{
public:
    explicit Grid(size_t inWidth = kDefaultWidth,
                  size_t inHeight = kDefaultHeight);
    Grid(const Grid<T*>& src);
    virtual ~Grid();
    Grid<T*>& operator=(const Grid<T*>& rhs);

    // The Grid becomes the owner of the element!
    void setElementAt(size_t x, size_t y, std::unique_ptr<T> inElem);
    T* getElementAt(size_t x, size_t y);
    const T* getElementAt(size_t x, size_t y) const;
```

```
size_t getHeight() const { return mHeight; }
size_t getWidth() const { return mWidth; }
static const size_t kDefaultWidth = 10;
static const size_t kDefaultHeight = 10;

private:
    void initializeCellsContainer();
    void copyFrom(const Grid<T*>& src);
    std::vector<std::vector<std::unique_ptr<T>>> mCells;
    size_t mWidth, mHeight;
};
```

As usual, these two lines are the crux of the matter:

```
template <typename T>
class Grid<T*>
```

The syntax says that this class is a specialization of the `Grid` template for all pointer types. You are providing the implementation only in cases where `T` is a pointer type. Note that if you instantiate a grid like this: `Grid<int*> myIntGrid`, then `T` will actually be `int`, not `int*`. That's a bit unintuitive, but unfortunately, the way it works. Here is a code example:

```
Grid<int> myIntGrid;           // Uses the non-specialized grid
Grid<int*> psGrid(2, 2); // Uses the partial specialization for pointer types

psGrid.setElementAt(0, 0, make_unique<int>(1));
psGrid.setElementAt(0, 1, make_unique<int>(2));
psGrid.setElementAt(1, 0, make_unique<int>(3));

Grid<int*> psGrid2(psGrid);
Grid<int*> psGrid3;
psGrid3 = psGrid2;

auto element = psGrid2.getElementAt(1, 0);
if (element) {
    cout << *element << endl;
    *element = 6;
}
cout << *psGrid.getElementAt(1, 0) << endl; // psGrid is not modified
cout << *psGrid2.getElementAt(1, 0) << endl; // psGrid2 is modified
```

The output is:

```
3
3
6
```

The implementations of the methods are rather straightforward, except for `copyFrom()`, which uses the copy constructor of individual elements to make a deep copy of them:

```
template <typename T>
void Grid<T*>::copyFrom(const Grid<T*>& src)
{
    mWidth = src.mWidth;
```

```

mHeight = src.mHeight;

initializeCellsContainer();

for (size_t i = 0; i < mWidth; i++) {
    for (size_t j = 0; j < mHeight; j++) {
        // Make a deep copy of the element by using its copy constructor.
        if (src.mCells[i][j]) {
            mCells[i][j].reset(new T(*src.mCells[i][j]));
        }
    }
}
}

```

EMULATING FUNCTION PARTIAL SPECIALIZATION WITH OVERLOADING

The C++ standard does not permit partial template specialization of functions. Instead, you can overload the function with another template. The difference is subtle. Suppose that you want to write a specialization of the `Find()` function template, presented in Chapter 11, that dereferences the pointers to use `operator==` directly on the objects pointed to. Following the syntax for class template partial specialization, you might be tempted to write this:

```

template <typename T>
size_t Find<T*>(T*& value, T** arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        if (*arr[i] == *value) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // failed to find it; return NOT_FOUND
}

```

However, that syntax declares a partial specialization of the function template, which the C++ standard does not allow. The correct way to implement the behavior you want is to write a new template for `Find()`. The difference might seem trivial and academic, but otherwise it won't compile.

```

template <typename T>
size_t Find(T*& value, T** arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        if (*arr[i] == *value) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // failed to find it; return NOT_FOUND
}

```

Note that the first parameter to this version of `Find()` is `T*&`. This is done to make it symmetric with the original `Find()` function template, which accepts a `T&` as a first parameter. However, in this case, using `T*` instead of `T*&` for the first parameter of the partial specialization of `Find()` works as well.

More on Deduction

You can define in one program the original `Find()` template, the overloaded `Find()` for partial specialization on pointer types, the complete specialization for `const char*`s, and the overloaded `Find()` just for `const char*`s. The compiler selects the appropriate version to call based on its deduction rules.

NOTE *Between all overloaded versions, function template specializations, and specific function template instantiations, the compiler always chooses the “most specific” one to call. If a non-template version is equally specific as a function template instantiation, then the compiler prefers the non-template version.*

The following code calls `Find()` several times. The comments say which version of `Find()` is called:

```
size_t res = NOT_FOUND;
int x = 3, intArr[] = {1, 2, 3, 4};
size_t sizeArr = sizeof(intArr) / sizeof(int);
res = Find(x, intArr, sizeArr);           // calls Find<int> by deduction
res = Find<int>(x, intArr, sizeArr); // calls Find<int> explicitly

double d1 = 5.6, dArr[] = {1.2, 3.4, 5.7, 7.5};
sizeArr = sizeof(dArr) / sizeof(double);
res = Find(d1, dArr, sizeArr);           // calls Find<double> by deduction
res = Find<double>(d1, dArr, sizeArr); // calls Find<double> explicitly

const char* word = "two";
const char* arr[] = {"one", "two", "three", "four"};
sizeArr = sizeof(arr) / sizeof(arr[0]);
// calls template specialization for const char*
res = Find<const char*>(word, arr, sizeArr);
res = Find(word, arr, sizeArr);           // calls overloaded Find for const char*

int *px = &x, *pArr[] = {&x, &x};
sizeArr = sizeof(pArr) / sizeof(pArr[0]);
res = Find(px, pArr, sizeArr);           // calls the overloaded Find for pointers

SpreadsheetCell c1(10), c2[] = {SpreadsheetCell(4), SpreadsheetCell(10)};
sizeArr = sizeof(c2) / sizeof(c2[0]);
res = Find(c1, c2, sizeArr);           // calls Find<SpreadsheetCell> by deduction
// calls Find<SpreadsheetCell> explicitly
res = Find<SpreadsheetCell>(c1, c2, sizeArr);

SpreadsheetCell *pc1 = &c1;
SpreadsheetCell *psa[] = {&c1, &c1};
sizeArr = sizeof(psa) / sizeof(psa[0]);
res = Find(pc1, psa, sizeArr);           // Calls the overloaded Find for pointers
```

TEMPLATE RECURSION

Templates in C++ provide capabilities that go far beyond the simple classes and functions you have seen so far in this chapter and Chapter 11. One of these capabilities is *template recursion*. This section first provides a motivation for template recursion, and then shows how to implement it.

This section uses operator overloading, discussed in Chapter 14. If you skipped that chapter or are unfamiliar with the syntax for overloading operator[], consult Chapter 14 before continuing.

An N-Dimensional Grid: First Attempt

The `Grid` template example up to now supports only two dimensions, which limits its usefulness. What if you wanted to write a 3-D Tic-Tac-Toe game or write a math program with four-dimensional matrices? You could, of course, write a templated or non-templated for each of those dimensions. However, that would repeat a lot of code. Another approach is to write only a single-dimensional grid. Then, you could create a `Grid` of any dimension by instantiating the `Grid` with another `Grid` as its element type. This `Grid` element type could itself be instantiated with a `Grid` as its element type, and so on. Here is the implementation of the `OneDGrid` class template. It's simply a one-dimensional version of the `Grid` template from earlier examples, with the addition of a `resize()` method, and the substitution of `operator[]` for `setElementAt()` and `getElementAt()`:

```
template <typename T>
class OneDGrid
{
public:
    explicit OneDGrid(size_t inSize = kDefaultSize);
    virtual ~OneDGrid();

    T& operator[](size_t x);
    const T& operator[](size_t x) const;

    void resize(size_t newSize);
    size_t getSize() const { return mElems.size(); }
    static const size_t kDefaultSize = 10;

private:
    std::vector<T> mElems;
};

template <typename T>
OneDGrid<T>::OneDGrid(size_t inSize)
{
    mElems.resize(inSize);
}

template <typename T>
OneDGrid<T>::~OneDGrid()
{
    // Nothing to do, the vector will clean up itself.
}

template <typename T>
```

```
void OneDGrid<T>::resize(size_t newSize)
{
    mElems.resize(newSize);
}

template <typename T>
T& OneDGrid<T>::operator[](size_t x)
{
    return mElems[x];
}

template <typename T>
const T& OneDGrid<T>::operator[](size_t x) const
{
    return mElems[x];
}
```

With this implementation of `OneDGrid`, you can create multidimensional grids like this:

```
OneDGrid<int> singleDGrid;
OneDGrid<OneDGrid<int>> twoDGrid;
OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;
singleDGrid[3] = 5;
twoDGrid[3][3] = 5;
threeDGrid[3][3][3] = 5;
```

This code works fine, but the declarations are messy. We can do better.

A Real N-Dimensional Grid

You can use template recursion to write a “real” N -dimensional grid because dimensionality of grids is essentially recursive. You can see that in this declaration:

```
OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;
```

You can think of each nesting `OneDGrid` as a recursive step, with the `OneDGrid` of `int` as the base case. In other words, a three-dimensional grid is a single-dimensional grid of single-dimensional grids of single-dimensional grids of `int`s. Instead of requiring the user to do this recursion, you can write a class template that does it for you. You can then create N -dimensional grids like this:

```
NDGrid<int, 1> singleDGrid;
NDGrid<int, 2> twoDGrid;
NDGrid<int, 3> threeDGrid;
```

The `NDGrid` class template takes a type for its element and an integer specifying its “dimensionality.” The key insight here is that the element type of the `NDGrid` is not the element type specified in the template parameter list, but is in fact another `NDGrid` of dimensionality one less than the current. In other words, a three-dimensional grid is a vector of two-dimensional grids; the two-dimensional grids are each vectors of one-dimensional grids.

With recursion, you need a base case. You can write a partial specialization of the `NDGrid` for dimensionality of 1, in which the element type is not another `NDGrid`, but is in fact the element type specified by the template parameter.

Here is the general `NDGrid` template definition, with highlights showing where it differs from the `OneDGrid` shown in the previous section:

```
template <typename T, size_t N>
class NDGrid
{
public:
    explicit NDGrid(size_t inSize = kDefaultSize);
    virtual ~NDGrid();

    NDGrid<T, N-1>& operator[](size_t x);
    const NDGrid<T, N-1>& operator[](size_t x) const;

    void resize(size_t newSize);
    size_t getSize() const { return mElems.size(); }
    static const size_t kDefaultSize = 10;

private:
    std::vector<NDGrid<T, N-1>> mElems;
};
```

Note that `mElems` is a vector of `NDGrid<T, N-1>`: This is the recursive step. Also, `operator[]` returns a reference to the element type, which is again `NDGrid<T, N-1>`, not `T`.

The template definition for the base case is a partial specialization for dimension 1:

```
template <typename T>
class NDGrid<T, 1>
{
public:
    explicit NDGrid(size_t inSize = kDefaultSize);
    virtual ~NDGrid();

    T& operator[](size_t x);
    const T& operator[](size_t x) const;

    void resize(size_t newSize);
    size_t getSize() const { return mElems.size(); }
    static const size_t kDefaultSize = 10;

private:
    std::vector<T> mElems;
};
```

Here the recursion ends: The element type is `T`, not another template instantiation.

The trickiest aspect of the implementations, other than the template recursion itself, is appropriately sizing each dimension of the grid. This implementation creates the N -dimensional grid with every dimension of equal size. It's significantly more difficult to specify a separate size for each dimension. However, even with this simplification, there is still a problem: The user should have the ability to create the array with a specified size, such as 20 or 50. Thus, the constructor takes an integer size parameter. However, when you dynamically resize a vector of sub-grids, you cannot pass this size value on to the sub-grid elements because vectors create objects using their default constructor.

Thus, you must explicitly call `resize()` on each grid element of the vector. That code follows. The base case doesn't need to resize its elements because the elements are `Ts`, not grids.

Here are the implementations of the main `NDGrid` template, with highlights showing the differences from the `OneDGrid`:

```
template <typename T, size_t N>
NDGrid<T, N>::NDGrid(size_t inSize)
{
    resize(inSize);
}

template <typename T, size_t N>
NDGrid<T, N>::~NDGrid()
{
    // Nothing to do, the vector will clean up itself.
}

template <typename T, size_t N>
void NDGrid<T, N>::resize(size_t newSize)
{
    mElems.resize(newSize);

    // Resizing the vector calls the 0-argument constructor for
    // the NDGrid<T, N-1> elements, which constructs
    // it with the default size. Thus, we must explicitly call
    // resize() on each of the elements to recursively resize all
    // nested Grid elements.
    for (auto& element : mElems) {
        element.resize(newSize);
    }
}

template <typename T, size_t N>
NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x)
{
    return mElems[x];
}

template <typename T, size_t N>
const NDGrid<T, N-1>& NDGrid<T, N>::operator[](size_t x) const
{
    return mElems[x];
}
```

Here are the implementations of the partial specialization (base case). Note that you must rewrite a lot of the code because you don't inherit any implementations with specializations. Highlights show the differences from the non-specialized `NDGrid`:

```
template <typename T>
NDGrid<T, 1>::NDGrid(size_t inSize)
{
    resize(inSize);
}
```

```

template <typename T>
NDGrid<T, 1>::~NDGrid()
{
    // Nothing to do, the vector will clean up itself.
}

template <typename T>
void NDGrid<T, 1>::resize(size_t newSize)
{
    mElems.resize(newSize);
}

template <typename T>
T& NDGrid<T, 1>::operator[](size_t x)
{
    return mElems[x];
}

template <typename T>
const T& NDGrid<T, 1>::operator[](size_t x) const
{
    return mElems[x];
}

```

Now, you can write code like this:

```

NDGrid<int, 3> my3DGrid;
my3DGrid[2][1][2] = 5;
my3DGrid[1][1][1] = 5;
cout << my3DGrid[2][1][2] << endl;

```

TYPE INFERENCE

Type inference is discussed in Chapter 1 and allows the compiler to automatically deduce the exact type of an expression. There are two keywords for type inference: `auto` and `decltype`. Type inference turns out to be very useful in combination with templates. This section goes into more detail on their use in a template context.

auto and decltype with Templates

The use of the `auto` and `decltype` keywords in combination with templates is best illustrated with an example. The following example defines two classes: `MyInt` and `MyString`. These are simple wrappers for an `int` and a `std::string`, respectively. Their constructors accept a single value used for initialization. Both classes also have an `operator+` as a member method. Here is the header file:

```

// Forward class declaration
class MyString;

class MyInt
{
public:
    MyInt(int i) : mValue(i) {}
    MyInt operator+(const MyString& rhs) const;

```

```
        int getInt() const { return mValue; }
private:
    int mValue;
};

class MyString
{
public:
    MyString(const std::string& str) : mString(str) {}
    MyString operator+(const MyInt& rhs) const;
    const std::string& getString() const { return mString; }
private:
    std::string mString;
};
```

The implementation is as follows. This code uses `stoi()` and `to_string()`, and both of them are discussed in the “Numeric Conversions” section in Chapter 2:

```
MyInt MyInt::operator+(const MyString& rhs) const
{
    return mValue + stoi(rhs.getString());
}
MyString MyString::operator+(const MyInt& rhs) const
{
    string str = mString;
    str.append(to_string(rhs.getInt()));
    return str;
}
```

These operators are implemented as such to force a different result depending on the order of the arguments to the addition operator. For example:

```
MyInt i(4);
MyString str("5");
MyInt a = i + str;
MyString b = str + i;
```

In this case, the type of variable `a` is `MyInt` and the type of variable `b` is `MyString`. Now imagine that you want to write a function template to perform the addition. You can write the following:

```
template<typename T1, typename T2, typename Result>
Result DoAddition(const T1& t1, const T2& t2)
{
    return t1 + t2;
}
```

As you can see, it requires you to specify three template parameters: the type of the first operand, the type of the second operand, and the type of the result of performing the addition. You can call this function template as follows. Note that you have to specify the three template type arguments. The compiler cannot deduce them from the supplied function arguments because you only supply two function arguments and there are three template type arguments.

```
auto c = DoAddition<MyInt, MyString, MyInt>(i, str);
```

This is obviously not that elegant because you need to manually specify the type of the return value. After reading about the `decltype` keyword, you might want to try to fix this issue as follows:

```
template<typename T1, typename T2>
decltype(t1 + t2) DoAddition(const T1& t1, const T2& t2)
{
    return t1 + t2;
}
```

However, this does not work because at the time of parsing the `decltype` keyword, the compiler doesn't know `t1` and `t2` yet. They become known further down in the function prototype. The correct solution is to combine the alternative function syntax with the `decltype` keyword as shown in the following implementation:

```
template<typename T1, typename T2>
auto DoAddition(const T1& t1, const T2& t2) -> decltype(t1 + t2)
{
    return t1 + t2;
}
```

With this implementation you can call `DoAddition()` as follows:

```
auto d = DoAddition(i, str);
auto e = DoAddition(str, i);
```

You can see that you do not need to specify any function template parameters anymore because the compiler can now deduce them based on the arguments given to `DoAddition()`, and can automatically figure out the type of the return value. In this example, `d` is of type `MyInt` and `e` is of type `MyString`.

With function return type deduction, the implementation of `DoAddition()` can be further simplified as follows:

```
template<typename T1, typename T2>
auto DoAddition(const T1& t1, const T2& t2)
{
    return t1 + t2;
}
```

VARIADIC TEMPLATES

Normal templates can take only a fixed number of template parameters. *Variadic templates* are templates that can take a variable number of template parameters. For example, the following code defines a template that can accept any number of template parameters, using a *parameter pack* called `Types`:

```
template<typename... Types>
class MyVariadicTemplate { };
```



NOTE The three dots behind typename are not an error. This is the syntax to define a parameter pack for variadic templates. A parameter pack is something that can accept a variable number of arguments. You are allowed to put spaces before and after the three dots.

You can instantiate `MyVariadicTemplate` with any number of types. For example:

```
MyVariadicTemplate<int> instance1;
MyVariadicTemplate<string, double, list<int>> instance2;
```

It can even be instantiated with zero template arguments:

```
MyVariadicTemplate<> instance3;
```

To avoid instantiating a variadic template with zero template arguments, you can write your template as follows:

```
template<typename T1, typename... Types>
class MyVariadicTemplate { };
```

With this definition, trying to instantiate `MyVariadicTemplate` with zero template arguments results in a compiler error. For example, with Microsoft Visual C++ you get the following error:

```
error C2976: 'MyVariadicTemplate' : too few template arguments
```

It is not possible to directly iterate over the different arguments given to a variadic template. The only way you can do this is with the aid of template recursion. The following sections show two examples of how to use variadic templates.

Type-Safe Variable-Length Argument Lists

Variadic templates allow you to create **type-safe variable-length** argument lists. The following example defines a variadic template called `processValues()` allowing it to accept a variable number of arguments with different types in a type-safe manner. The `processValues()` function processes each value in the variable-length argument list and executes a function called `handleValue()` for each single argument. This means that you have to write a `handleValue()` function for each type that you want to handle; `int`, `double` and `string` in this example:

```
void handleValue(int value) { cout << "Integer: " << value << endl; }
void handleValue(double value) { cout << "Double: " << value << endl; }
void handleValue(const string& value) { cout << "String: " << value << endl; }

template<typename T>
void processValues(T arg) // Base case
{
    handleValue(arg);
}

template<typename T1, typename... Tn>
void processValues(T1 arg1, Tn... args)
```

```

{
    handleValue(arg1);
    processValues(args...);
}

```

What this example also demonstrates is the double use of the triple dots ... operator. This operator appears in three places and has two different meanings. First, it is used behind `typename` in the template parameter list and behind type `Tn` in the function parameter list. In both cases it denotes a *parameter pack*. A parameter pack can accept a variable number of arguments.

The second type of use of the ... operator is behind the parameter name `args` in the function body. In this case, it means a *parameter pack expansion*; The operator *unpacks/expands* the parameter pack into separate arguments. It basically takes what is on the left of the operator, repeats it for every template parameter in the pack, separated by commas. Take the following line:

```
processValues(args...);
```

This line unpacks/expands the `args` parameter pack into its separate arguments, separated by commas, and then calls the `processValues()` function with the list of expanded arguments.

The template always requires at least one template parameter, `T1`. The act of recursively calling `processValues()` with `args...` is that on each call there is one template parameter less.

Because the implementation of the `processValues()` function is recursive, you need to have a way to stop the recursion. This is done by implementing a `processValues()` function template that accepts just a single template argument.

You can test the `processValues()` variadic template as follows:

```
processValues(1, 2, 3.56, "test", 1.1f);
```

The recursive calls generated by this example are as follows:

```

processValues(1, 2, 3.56, "test", 1.1f);
    handleValue(1);
    processValues(2, 3.56, "test", 1.1f);
        handleValue(2);
        processValues(3.56, "test", 1.1f);
            handleValue(3.56);
            processValues("test", 1.1f);
                handleValue("test");
                processValues(1.1f);
                    handleValue(1.1f);

```

It is important to remember that this method of variable-length argument lists is fully type-safe. The `processValues()` function automatically calls the correct `handleValue()` overload based on the actual type. Automatic casting can happen as usual in C++. For example, the `1.1f` in the preceding example is of type `float`. The `processValues()` function calls `handleValue(double value)` because conversion from `float` to `double` is without any loss. However, the compiler will issue an error when you call `processValues()` with an argument of a certain type for which there is no `handleValue()` defined.

There is a problem though with the preceding implementation. Because it's a recursive implementation, the parameters are copied for each recursive call to `processValues()`. This can become costly depending on the type of the arguments. You might think that you can avoid this copying by passing references to `processValues()` instead of using pass-by-value. Unfortunately that also means that you cannot call `processValues()` with literals anymore, because a reference to a literal value is not allowed, unless you use `const` references.

To use non-`const` references and still allow literal values, you can use rvalue references, discussed in Chapter 10. The following implementation uses rvalue references, and uses `std::forward()` to *perfectly forward* all parameters. If an rvalue reference is passed to `processValues()`, then it is forwarded as an rvalue reference. If an lvalue reference is passed, then it is forwarded as an lvalue reference.

```
template<typename T>
void processValues(T&& arg)
{
    handleValue(std::forward<T>(arg));
}

template<typename T1, typename... Tn>
void processValues(T1&& arg1, Tn&&... args)
{
    handleValue(std::forward<T1>(arg1));
    processValues(std::forward<Tn>(args)...);
}
```

There is one line that needs further explanation:

```
    processValues(std::forward<Tn>(args)...);
```

The `...` operator is used to unpack the parameter pack. It uses `std::forward()` on each individual argument in the pack and separates them with commas.

Inside the body of a function using a parameter pack you can retrieve the number of arguments in the pack as follows:

```
int numArgs = sizeof...(args);
```

A practical example of using variadic templates is to write a secure and type-safe version of `printf()`. This would be a good exercise to practice variadic templates.

Variable Number of Mixin Classes

Parameter packs can be used almost everywhere. For example, the following code uses a parameter pack to define a variable number of mixin classes for `MyClass`. Chapter 5 discusses the concept of mixin classes.

```
class Mixin1
{
public:
    Mixin1(int i) : mValue(i) {}
    virtual void Mixin1Func() { cout << "Mixin1: " << mValue << endl; }
private:
```

```

        int mValue;
    };

    class Mixin2
    {
    public:
        Mixin2(int i) : mValue(i) {}
        virtual void Mixin2Func() { cout << "Mixin2: " << mValue << endl; }
    private:
        int mValue;
    };

    template<typename... Mixins>
    class MyClass : public Mixins...
    {
    public:
        MyClass(const Mixins&... mixins) : Mixins(mixins)... {}
        virtual ~MyClass() {}
    };

```

This code first defines two mixin classes: `Mixin1` and `Mixin2`. They are kept pretty simple for this example. Their constructor accepts an integer, which is stored, and they have a function to print information about that specific instance of the class. The `MyClass` variadic template uses a parameter pack `typename... Mixins` to accept a variable number of mixin classes. This implementation requires you to specify at least one mixin class. The class then inherits from all those mixin classes and the constructor accepts the same number of arguments to initialize each inherited mixin class. Remember that the `...` expansion operator basically takes what is on the left of the operator and repeats it for every template parameter in the pack, separated by commas. The class can be used as follows:

```

MyClass<Mixin1, Mixin2> a(Mixin1(11), Mixin2(22));
a.Mixin1Func();
a.Mixin2Func();

MyClass<Mixin1> b(Mixin1(33));
b.Mixin1Func();
//b.Mixin2Func(); // Error: does not compile.

```

When you try to call `Mixin2Func()` on `b` you will get a compiler error because `b` is not inheriting from the `Mixin2` class. The output of this program is as follows:

```

Mixin1: 11
Mixin2: 22
Mixin1: 33

```

METaproGRAMMING

This section touches on *template metaprogramming*. It is a very complicated subject and there are books written about it explaining all the little details. This book doesn't have the space to go into all the details of metaprogramming. Instead, this section explains the most important concepts, with the aid of a couple of examples.

The goal of template metaprogramming is to perform some computation at compile time instead of at run time. It is basically a programming language on top of C++. The following section starts the discussion with a simple example that calculates the factorial of a number at compile time and makes the result available as a simple constant at run time.

Factorial at Compile Time

Template metaprogramming allows you to perform calculations at compile time instead of at run time. The following code is a small example that calculates the factorial of a number at compile time. The code uses template recursion, explained earlier in this chapter, which requires a recursive template and a base template to stop the recursion. By mathematical definition, the factorial of 0 is 1, so that is used as the base case.

```
template<unsigned char f>
class Factorial
{
public:
    static const unsigned long long val = (f * Factorial<f - 1>::val);
};

template<>
class Factorial<0>
{
public:
    static const unsigned long long val = 1;
};

int main()
{
    cout << Factorial<6>::val << endl;
    return 0;
}
```

This calculates the factorial of 6, mathematically written as $6!$, which is $1 \times 2 \times 3 \times 4 \times 5 \times 6$ or 720.

NOTE *It is important to keep in mind that the factorial calculation is happening at compile time. At run time you simply access the compile-time calculated value through `::val`, which is just a static constant value.*

Loop Unrolling

A second example of template metaprogramming is to unroll loops at compile time instead of executing the loop at run time. Note that loop unrolling should only be done when you really need it because usually the compiler is smart enough to unroll loops that can be unrolled for you.

This example again uses template recursion because it needs to do something in a loop at compile time. For template recursion you need the recursive implementation of the template and a base

template that stops the recursion. On each recursion, the `Loop` template instantiates itself with `i-1`. When it hits 0, the recursion stops.

```
template<int i>
class Loop
{
public:
    template <typename FuncType>
    static inline void Do(FuncType func) {
        Loop<i - 1>::Do(func);
        func(i);
    }
};

template<>
class Loop<0>
{
public:
    template <typename FuncType>
    static inline void Do(FuncType /* func */) { }
};
```

The `Loop` template can be used as follows:

```
void DoWork(int i) { cout << "DoWork(" << i << ")" << endl; }
int main()
{
    Loop<3>::Do(DoWork);
}
```

This code causes the compiler to unroll the loop and to call the function `DoWork()` three times in a row. The output of the program is:

```
DoWork(1)
DoWork(2)
DoWork(3)
```

Using `std::bind()` you can use a version of `DoWork()` that accepts more than one parameter:

```
void DoWork2(string str, int i)
{
    cout << "DoWork2(" << str << ", " << i << ")" << endl;
}
int main()
{
    Loop<2>::Do(bind(DoWork2, "TestStr", placeholders::_1));
}
```

The code first implements a function that accepts a `string` and an `int`. The `main()` function uses `std::bind()` to bind the first parameter of `DoWork2()` to a fixed string, "TestStr". See Chapter 17 for details on `std::bind()`. If you compile and run this code, the output is as follows:

```
DoWork2(TestStr, 1)
DoWork2(TestStr, 2)
```

Printing Tuples

This example uses template metaprogramming to print the individual elements of an `std::tuple`. Tuples are explained in Chapter 19. They allow you to store any number of values, each with its own specific type. A tuple has a fixed size and fixed value types, determined at compile time. However, tuples don't have any built-in mechanism to iterate over their elements. The following example shows how you could use template metaprogramming to iterate over the elements of a tuple at compile time.

As is often the case with template metaprogramming, this example is again using template recursion. The `tuple_print` class template has two template parameters: the `tuple` type, and an integer, initialized with the size of the tuple. It then recursively instantiates itself in the constructor and decrements the integer on every call. A partial specialization of `tuple_print` stops the recursion when this integer hits 0. The `main()` function shows how this `tuple_print` class template can be used.

```
template<int n, typename TupleType>
class tuple_print
{
public:
    tuple_print(const TupleType& t) {
        tuple_print<n - 1, TupleType> tp(t);
        cout << get<n - 1>(t) << endl;
    }
};

template<typename TupleType>
class tuple_print<0, TupleType>
{
public:
    tuple_print(const TupleType&) { }
};

int main()
{
    using MyTuple = tuple<int, string, bool>;
    MyTuple t1(16, "Test", true);
    tuple_print<tuple_size<MyTuple>::value, MyTuple> tp(t1);
}
```

If you look at the `main()` function, you can see that the line to use the `tuple_print` template looks a bit complicated because it requires the size of the tuple and the exact type of the tuple as template arguments. This can be simplified a lot by introducing a helper function template that automatically deduces the template parameters. The simplified implementation is as follows:

```
template<int n, typename TupleType>
class tuple_print_helper
{
public:
    tuple_print_helper(const TupleType& t) {
        tuple_print_helper<n - 1, TupleType> tp(t);
        cout << get<n - 1>(t) << endl;
    }
};
```

```

        }
    };

template<typename TupleType>
class tuple_print_helper<0, TupleType>
{
public:
    tuple_print_helper(const TupleType&) { }
};

template<typename T>
void tuple_print(const T& t)
{
    tuple_print_helper<tuple_size<T>::value, T> tph(t);
}

int main()
{
    auto t1 = make_tuple(167, "Testing", false, 2.3);
    tuple_print(t1);
}

```

The first change made here is renaming the original `tuple_print` class template to `tuple_print_helper`. The code then implements a small function template called `tuple_print()`, which accepts the type of the tuple as a template argument and accepts a reference to the tuple itself as a function argument. The body of that function instantiates the `tuple_print_helper` class template. The `main()` function shows how to use this simplified version. Because you don't need to know the exact type of the tuple yourself anymore, you can use the recommended `make_tuple()` together with the `auto` keyword to avoid having to write the tuple type yourself. The call to the `tuple_print()` function template is very simple:

```
tuple_print(t1);
```

You don't need to specify the function template argument because the compiler can deduce this automatically from the supplied argument.

Type Traits

Type traits allow you to make decisions based on types at compile time. For example, you can write a template that requires a type that is derived from a certain type, or a type that is convertible to a certain type, or a type that is integral, and so on. The standard defines several helper classes for this. All type traits-related functionality is defined in the `<type_traits>` header file. The following list gives a few examples of the available type traits-related classes in the standard. Consult a Standard Library Reference — for example <http://www.cppreference.com/> or <http://www.cplusplus.com/> reference — for a complete list.

<ul style="list-style-type: none"> ➤ Primary type categories <ul style="list-style-type: none"> is_void is_integral is_floating_point is_pointer ... ➤ Type properties <ul style="list-style-type: none"> is_const is_literal_type is_polymorphic is_unsigned is_constructible is_copy_constructible is_move_constructible is_assignable is_trivially_copyable ... ➤ Reference modifications <ul style="list-style-type: none"> remove_reference add_lvalue_reference add_rvalue_reference 	<ul style="list-style-type: none"> ➤ Composed type categories <ul style="list-style-type: none"> is_reference is_object is_scalar ... ➤ Type relations <ul style="list-style-type: none"> is_same is_base_of is_convertible ➤ const-volatile modifications <ul style="list-style-type: none"> remove_const add_const ... ➤ Sign modifications <ul style="list-style-type: none"> make_signed make_unsigned ➤ Other transformations <ul style="list-style-type: none"> enable_if conditional ...
--	--

Type traits is a pretty advanced C++ feature. By just looking at the preceding list, which is already a shortened version of the list from the standard itself, it is clear that this book cannot explain all details about all type traits. This section explains just a couple of use cases to show you how type traits can be used.

Using Type Categories

Before an example can be given for a template using type traits, you first need to know a bit more on how classes like `is_integral` work. The standard defines an `integral_constant` class that looks as follows:

```
template <class T, T v>
struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
```

```

typedef integral_constant<T,v> type;
constexpr operator value_type() const noexcept { return value; }
constexpr value_type operator()() const noexcept { return value; }
};

typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

```

What this defines is two types: `true_type` and `false_type`. When you call `true_type::value` you get the value `true` and when you call `false_type::value` you get the value `false`. You can also call `true_type::type`, which returns the type of `true_type`. The same holds for `false_type`. Classes like `is_integral` and `is_class` inherit from `integral_constant`. For example, `is_integral` can be specialized for type `bool` as follows:

```
template<> struct is_integral<bool> : public true_type { };
```

This allows you to write `is_integral<bool>::value`, which returns the value `true`. Note that you don't need to write these specializations yourself; they are part of the standard library.

The following code shows the simplest example of how type categories can be used:

```

if (is_integral<int>::value) {
    cout << "int is integral" << endl;
} else {
    cout << "int is not integral" << endl;
}
if (is_class<string>::value) {
    cout << "string is a class" << endl;
} else {
    cout << "string is not a class" << endl;
}

```

This example is using `is_integral` to check whether `int` is an integral type or not, and uses `is_class` to check whether `string` is a class or not. The output is as follows:

```
int is integral
string is a class
```

Of course, you will likely never use type traits in this way. They become more useful in combination with templates to generate code based on some properties of a type. The following template example demonstrates this. The code defines an overloaded function template `process_helper()` that accepts a type as template argument. The first argument to this function is a value, and the second argument is either an instance of `true_type` or `false_type`. The `process()` function template accepts a single argument and calls `process_helper()`:

```

template<typename T>
void process_helper(const T& t, true_type)
{
    cout << t << " is an integral type." << endl;
}

template<typename T>
void process_helper(const T& t, false_type)

```

```
{  
    cout << t << " is a non-integral type." << endl;  
}  
  
template<typename T>  
void process(const T& t)  
{  
    process_helper(t, typename is_integral<T>::type());  
}
```

The second parameter in the call to `process_helper()` is:

```
typename is_integral<T>::type()
```

This uses `is_integral` to figure out if `T` is an integral type. `::type` returns the resulting `integral_constant` type, which can be `true_type` or `false_type`. The `process_helper()` function needs an instance of `true_type` or `false_type` as a second parameter, so that is the reason for the two empty parentheses behind `::type`. Note that the two overloaded `process_helper()` functions use nameless parameters of type `true_type` and `false_type`. They are nameless because they don't use those parameters inside their function body. These parameters are only used for function overload resolution.

The code can be tested as follows:

```
process(123);  
process(2.2);  
process(string("Test"));
```

The output of this example is as follows:

```
123 is an integral type.  
2.2 is a non-integral type.  
Test is a non-integral type.
```

The previous example could be written using only a `process()` function template as follows, but that doesn't demonstrate how to write different overloads where one parameter is either `true_type` or `false_type`.

```
template<typename T>  
void process(const T& t)  
{  
    if (is_integral<T>::value) {  
        cout << t << " is an integral type." << endl;  
    } else {  
        cout << t << " is a non-integral type." << endl;  
    }  
}
```

Using Type Relations

There are three type relations available: `is_same`, `is_base_of`, and `is_convertible`. This section gives an example on how to use `is_same`; the others work similarly.

The following `same()` function template uses the `is_same` type trait to figure out whether the two given arguments are of the same type or not, and outputs an appropriate message. Using the `same()` function template is very easy as is shown in the `main()` function:

```
template<typename T1, typename T2>
void same(const T1& t1, const T2& t2)
{
    bool areTypesTheSame = is_same<T1, T2>::value;
    cout << "" << t1 << " and " << t2 << "' are ";
    cout << (areTypesTheSame ? "the same types." : "different types.") << endl;
}

int main()
{
    same(1, 32);
    same(1, 3.01);
    same(3.01, string("Test"));
}
```

The output is as follows:

```
'1' and '32' are the same types.
'1' and '3.01' are different types
'3.01' and 'Test' are different types
```

Using `enable_if`

First a little warning before continuing with `enable_if`. Using `enable_if` requires knowledge of a feature called *Substitution Failure Is Not An Error* (SFINAE), a complicated and obscure feature of C++. This section explains the basics of SFINAE with an example using `enable_if`.

If you have a set of overloaded functions, you can use `enable_if` to selectively disable certain overloads based on some type traits. `enable_if` is usually used on the return types for your set of overloads. `enable_if` accepts two template type parameters. The first is a Boolean, and the second is a type that is `void` by default. If the Boolean is `true`, then the `enable_if` class has a nested type that you can access using `::type`. The type of this nested type is the type given as a second template type parameter. If the Boolean is `false`, then there is no nested type.

The example from the previous section with the `same()` function template can be rewritten into an overloaded `check_type()` function template by using `enable_if` as follows. In this version, the `check_type()` functions return `true` or `false` depending on whether the types of the given values are the same or not. If you don't want to return anything from `check_type()` you can remove the second template type parameter for `enable_if` or replace it with `void`.

```
template<typename T1, typename T2>
typename enable_if<is_same<T1, T2>::value, bool>::type
    check_type(const T1& t1, const T2& t2)
{
    cout << "" << t1 << " and " << t2 << "' ";
    cout << "are the same types." << endl;
    return true;
}
```

```
template<typename T1, typename T2>
typename enable_if<!is_same<T1, T2>::value, bool>::type
    check_type(const T1& t1, const T2& t2)
{
    cout << "'" << t1 << "' and '" << t2 << "' ";
    cout << "are different types." << endl;
    return false;
}

int main()
{
    check_type(1, 32);
    check_type(1, 3.01);
    check_type(3.01, string("Test"));
}
```

The output is the same as before:

```
'1' and '32' are the same types.
'1' and '3.01' are different types.
'3.01' and 'Test' are different types.
```

The code defines two versions of `check_type()`. The return type of both versions is the nested type of `enable_if`. First, `is_same` is used to check whether the two types are the same or not and the result is retrieved with `::value`. This value is given to `enable_if`, and `::type` is used to get the resulting type, which is specified to be of type `bool`. When the first argument to `enable_if` is `true` then `::type` has type `bool`. When the first argument to `enable_if` is `false`, then there is no wrapped type, so `::type` fails. This is where SFINAE comes into play.

When the compiler starts to compile the first line of `main()`, it tries to find a function `check_type()` that accepts two integer values. It finds the first `check_type()` function template overload in the source code and deduces that it can use an instance of this by making `T1` and `T2` both integers. It then tries to figure out the return type. Since both arguments are integers and thus the same types, `is_same<T1, T2>::value` returns `true`, which causes `enable_if<true, bool>::type` to be of type `bool`. With this instantiation, everything is fine and the compiler can use that version of `check_type()`.

However, when the compiler tries to compile the second line in `main()`, it again tries to find a suitable `check_type()` function. It starts with the first `check_type()` and decides it can use that overload by setting `T1` to type `integer` and `T2` to type `double`. It then tries to figure out the return type. This time, `T1` and `T2` are different types, which means that `is_same<T1, T2>::value` returns `false`. Because of this, `enable_if<false, bool>` has no wrapped type and thus `enable_if<false, bool>::type` fails, leaving the function `check_type()` without a return type. The compiler notices this error but does not yet generate a real compilation error because of SFINAE. The compiler gracefully backtracks and tries to find another `check_type()` function. In this case the second `check_type()` works out perfectly because `!is_same<T1, T2>::value` is `true` and thus `enable_if<true, bool>::type` is of type `bool`.

If you want to use `enable_if` on a set of constructors, you can't use it for the return type because constructors don't have a return type. In that case, you can use `enable_if` on an extra constructor parameter with a default value.

It is recommended to use `enable_if` judiciously. Use it only when you need to resolve overload ambiguities that you cannot possibly resolve using any other technique such as specialization, partial specialization, and so on. For example, if you just want compilation to fail when you use a template with the wrong types, use `static_assert()`, explained in Chapter 26, and not SFINAE. Of course there are legitimate use cases for `enable_if`. One such example is specializing a copy function for a custom vector-like class to perform bit-wise copying (for example, with the C function `memcpy()`) of trivially copyable types using `enable_if` and the `is_trivially_copyable` type trait.

WARNING *Relying on SFINAE is tricky and complicated. If your use of SFINAE and enable_if selectively disables the wrong overloads in your overload set, you will get cryptic compiler errors, which will be hard to track down.*

Metaprogramming Conclusion

As you have seen in this section, template metaprogramming can be a very powerful tool, but it can also get quite complicated. One problem with template metaprogramming, not yet mentioned before, is that everything happens at compile time so you cannot use a debugger to pinpoint a problem. If you decide to use template metaprogramming in your code, make sure you write good comments to explain exactly what is going on and why you are doing something a certain way. If you don't properly document your template metaprogramming code, it might be very difficult for someone else to understand your code, and it might even make it difficult for yourself to understand your own code in the future.

SUMMARY

This chapter is a continuation of the template discussion from Chapter 11. These chapters show you how to use templates for generic programming, and template metaprogramming for compile-time computations. Hopefully you gained an appreciation for the power and capabilities of these features, and an idea of how you could apply these concepts to your own code. Don't worry if you didn't understand all the syntax, or follow all the examples, on your first reading. The concepts can be difficult to grasp when you are first exposed to them, and the syntax is tricky whenever you want to write somewhat more complicated templates. When you actually sit down to write a class or function template, you can consult this chapter and Chapter 11 for a reference on the proper syntax.

22

Memory Management

WHAT'S IN THIS CHAPTER?

- Different ways to use and manage memory
- The often-perplexing relationship between arrays and pointers
- A low-level look at working with memory
- Smart pointers and how to use them
- Solutions to a few memory-related problems

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

In many ways, programming in C++ is like driving without a road. Sure, you can go anywhere you want, but there are no lines or traffic lights to keep you from injuring yourself. C++, like the C language, has a hands-off approach toward its programmers. The language assumes that you know what you're doing. It allows you to do things that are likely to cause problems because C++ is incredibly flexible and sacrifices safety in favor of performance.

Memory allocation and management is a particularly error-prone area of C++ programming. To write high-quality C++ programs, professional C++ programmers need to understand how memory works behind the scenes. This chapter explores the ins and outs of memory management. You will learn about the pitfalls of dynamic memory and some techniques for avoiding and eliminating them.

This chapter comes late in the book because in modern C++ you should avoid low-level memory operations as much as possible. For example, instead of dynamically allocated C-style arrays you should use STL containers, such as `vector`, which handle all memory management automatically for you. Instead of naked pointers you should use smart pointers, such as `unique_ptr` and `shared_ptr`, which automatically free the underlying resource, such as memory, when it's not needed anymore. Basically, you should try to avoid having calls to memory allocation routines such as `new` and `delete` in your code. Of course, it might not always be possible, and in existing code it will most likely not be the case, so a professional C++ programmer still needs to know how memory works behind the scenes.

WARNING *In modern C++ you should avoid low-level memory operations as much as possible in favor of modern constructs such as containers and smart pointers.*

WORKING WITH DYNAMIC MEMORY

Memory is a low-level component of the computer that sometimes unfortunately rears its head even in a high-level programming language like C++. Many programmers understand only enough about dynamic memory to get by. They shy away from data structures that use dynamic memory, or get their programs to work by trial and error. A solid understanding of how dynamic memory really works in C++ is essential to becoming a professional C++ programmer.

How to Picture Memory

Understanding dynamic memory is much easier if you have a mental model for what objects look like in memory. In this book, a unit of memory is shown as a box with a label next to it. The label indicates a variable name that corresponds to the memory. The data inside the box displays the current value of the memory.

For example, Figure 22-1 shows the state of memory after the following line is executed. The line should be in a function, so that `i` is a local variable:

```
int i = 7;
```

Since `i` is a local variable, it is allocated on the stack because it is declared as a simple type, not dynamically using the `new` keyword.



FIGURE 22-1

When you use the `new` keyword, memory is allocated on the heap. The following code creates a variable `ptr` on the stack, and then allocates memory on the heap to which `ptr` points.

```
int* ptr;
ptr = new int;
```

Figure 22-2 shows the state of memory after this code is executed. Notice that the variable `ptr` is still on the stack even though it points to memory on the heap. A pointer is just a variable and can live either on the stack or the heap, although this fact is easy to forget. Dynamic memory, however, is always allocated on the heap.



FIGURE 22-2

The next example shows that pointers can exist both on the stack and on the heap:

```
int** handle;
handle = new int*;
*handle = new int;
```

The preceding code first declares a pointer to a pointer to an integer as the variable `handle`. It then dynamically allocates enough memory to hold a pointer to an integer, storing the pointer to that new memory in `handle`. Next, that memory (`*handle`) is assigned a pointer to another section of dynamic memory that is big enough to hold the integer. Figure 22-3 shows the two levels of pointers with one pointer residing on the stack (`handle`) and the other residing on the heap (`*handle`).

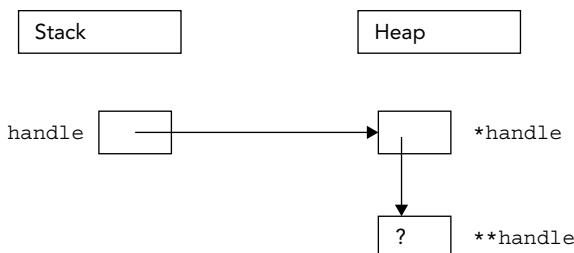


FIGURE 22-3

Allocation and Deallocation

To create space for a variable, you use the `new` keyword. To release that space for use by other parts of the program, you use the `delete` keyword. Of course, it wouldn't be C++ if simple concepts such as `new` and `delete` didn't have several variations and intricacies.

Using new and delete

When you want to allocate a block of memory, you call `new` with the type of variable for which you need space. `new` returns a pointer to that memory, although it is up to you to store that pointer in a variable. If you ignore the return value of `new`, or if the pointer variable goes out of scope, the memory becomes *orphaned* because you no longer have a way to access it. This is also called a *memory leak*.

For example, the following code orphans enough memory to hold an `int`. Figure 22-4 shows the state of memory after the code is executed. When there are blocks of data on the heap with no access, direct or indirect, from the stack, the memory is orphaned or leaked.

```
void leaky()
{
    new int;    // BUG! Orphans memory!
    cout << "I just leaked an int!" << endl;
}
```

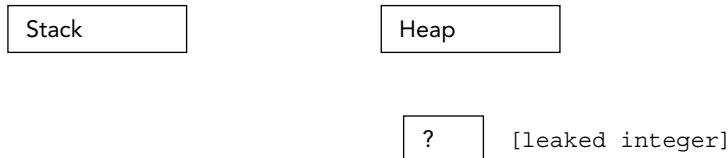


FIGURE 22-4

Until they find a way to make computers with an infinite supply of fast memory, you will need to tell the compiler when the memory associated with an object can be released and used for another purpose. To free memory on the heap, use the `delete` keyword with a pointer to the memory, as shown here:

```
int* ptr;
ptr = new int;
delete ptr;
```

WARNING *As a rule of thumb, every line of code that allocates memory with new and that uses a naked pointer instead of storing the pointer in a smart pointer, should correspond to another line of code that releases the same memory with delete.*

What about My Good Friend malloc?

If you are a C programmer, you may be wondering what is wrong with the `malloc()` function. In C, `malloc()` is used to allocate a given number of bytes of memory. For the most part, using `malloc()` is simple and straightforward. The `malloc()` function still exists in C++, but you should avoid it. The main advantage of `new` over `malloc()` is that `new` doesn't just allocate memory, it constructs objects!

For example, consider the following two lines of code, which use a hypothetical class called `Foo`:

```
Foo* myFoo = (Foo*)malloc(sizeof(Foo));
Foo* myOtherFoo = new Foo();
```

After executing these lines, both `myFoo` and `myOtherFoo` will point to areas of memory on the heap that are big enough for a `Foo` object. Data members and methods of `Foo` can be accessed using both pointers. The difference is that the `Foo` object pointed to by `myFoo` isn't a proper object because it was never constructed. The `malloc()` function only sets aside a piece of memory of a certain size. It doesn't know about or care about objects. In contrast, the call to `new` allocates the appropriate size of memory and also calls an appropriate constructor to construct the object. Chapter 14 describes these two duties of `new` in more detail.

A similar difference exists between the `free()` function and the `delete` operator. With `free()`, the object's destructor is not called. With `delete`, the destructor is called and the object is properly cleaned up.

WARNING *You should never use `malloc()` and `free()` in C++. Use only `new` and `delete`.*

When Memory Allocation Fails

Many, if not most, programmers write code with the assumption that `new` will always be successful. The rationale is that if `new` fails, it means that memory is very low and life is very, very bad. It is often an unfathomable state to be in because it's unclear what your program could possibly do in this situation.

By default, your program will terminate if `new` fails. In many programs, this behavior is acceptable. The program exits when `new` fails because `new` throws an exception if there is not enough memory available for the request. Chapter 13 explains approaches to recover gracefully from an out-of-memory situation.

There is also an alternative version of `new`, which will not throw an exception. Instead, it will return `nullptr`, similar to the behavior of `malloc()` in C. The syntax for using this version is shown here:

```
int* ptr = new(nothrow) int;
```

Of course, you still have the same problem as the version that throws an exception — what do you do when the result is `nullptr`? The compiler doesn't require you to check the result, so the `nothrow` version of `new` is more likely to lead to other bugs than the version that throws an exception. For this reason, it's suggested that you use the standard version of `new`. If out-of-memory recovery is important to your program, the techniques covered in Chapter 13 give you all the tools you need.

Arrays

Arrays package multiple variables of the same type into a single variable with indices. Working with arrays quickly becomes natural to a novice programmer because it is easy to think about values in numbered slots. The in-memory representation of an array is not far off from this mental model.

Arrays of Basic Types

When your program allocates memory for an array, it is allocating *contiguous* pieces of memory, where each piece is large enough to hold a single element of the array. For example, a local array of five ints can be declared on the stack as follows:

```
int myArray[5];
```

Figure 22-5 shows the state of memory after the array is created. When creating arrays on the stack, the size must be a constant value known at compile time.

NOTE *Some compilers allow variable-sized arrays on the stack. This is not a standard feature of C++, so I recommend cautiously backing away when you see it.*

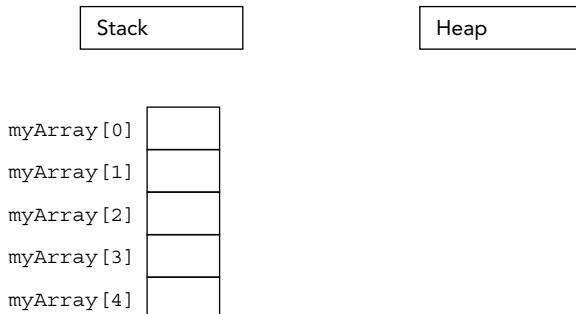


FIGURE 22-5

Declaring arrays on the heap is no different, except that you use a pointer to refer to the location of the array. The following code allocates memory for an array of five ints and stores a pointer to the memory in a variable called `myArrayPtr`.

```
int* myArrayPtr = new int[5];
```

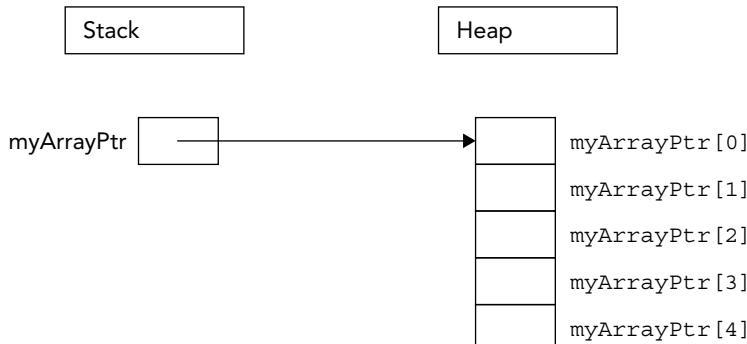


FIGURE 22-6

As Figure 22-6 illustrates, the heap-based array is similar to the stack-based array, but in a different location. The `myArrayPtr` variable points to the 0th element of the array.

Each call to `new[]` should be paired with a call to `delete[]` to clean up the memory. For example:

```
delete [] myArrayPtr;
```

The advantage of putting an array on the heap is that you can use dynamic memory to define its size at run time. For example, the following function receives a desired number of documents from a hypothetical function named `askUserForNumberOfDocuments()` and uses that result to create an array of `Document` objects.

```
Document* createDocArray()
{
    int numDocs = askUserForNumberOfDocuments();
    Document* docArray = new Document [numDocs];
    return docArray;
}
```

Remember that each call to `new[]` should be paired with a call to `delete[]`, so in this example, it's important that the caller of `createDocArray()` uses `delete[]` to clean up the returned memory. Another problem is that C-style arrays don't know their size, thus callers of `createDocArray()` have no idea how many elements there are in the returned array.

In the preceding function, `docArray` is a dynamically allocated array. Do not get this confused with a *dynamic array*. The array itself is not dynamic because its size does not change once it is allocated. Dynamic memory lets you specify the size of an allocated block at run time, but it does not automatically adjust its size to accommodate the data.

NOTE *There are data structures, such as STL containers, that do dynamically adjust their size and that do know their actual size. You should use these STL containers instead of standard arrays because they are much safer to use.*

There is a function in C++ called `realloc()`, which is a holdover from the C language. Don't use it! In C, `realloc()` is used to effectively change the size of an array by allocating a new block of memory of the new size, copying all of the old data to the new location, and deleting the original block. This approach is extremely dangerous in C++ because user-defined objects will not respond well to bitwise copying.

WARNING *Do not use `realloc()` in C++. It is not your friend.*

Arrays of Objects

Arrays of objects are no different than arrays of simple types. When you use `new[N]` to allocate an array of N objects, enough space is allocated for N contiguous blocks where each block is large enough for a single object. Using `new[]`, the zero-argument constructor for each of the objects will

automatically be called. In this way, allocating an array of objects using `new[]` will return a pointer to an array of fully formed and initialized objects.

For example, consider the following class:

```
class Simple
{
public:
    Simple() { cout << "Simple constructor called!" << endl; }
    virtual ~Simple() { cout << "Simple destructor called!" << endl; }
};
```

If you allocate an array of four `Simple` objects, the `Simple` constructor is called four times.

```
Simple* mySimpleArray = new Simple[4];
```

The memory diagram for this array is shown in Figure 22-7. As you can see, it is no different than an array of basic types.

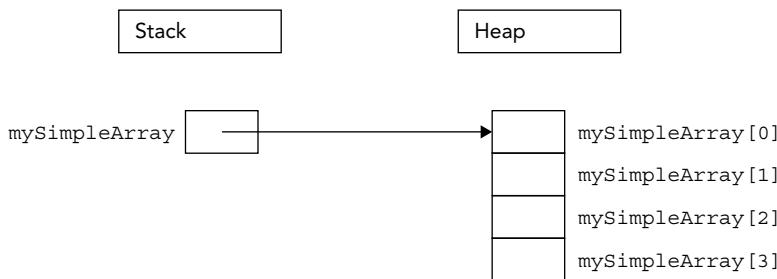


FIGURE 22-7

Deleting Arrays

As mentioned earlier, when you allocate memory with the array version of `new` (`new[]`), you must release it with the array version of `delete` (`delete[]`). This version will automatically destruct the objects in the array in addition to releasing the memory associated with them. If you do not use the array version of `delete`, your program may behave in odd ways. In some compilers, only the destructor for the 0th element of the array will be called because the compiler only knows that you are deleting a pointer to an object, and all the other elements of the array will become orphaned objects. In other compilers, memory corruption may occur because `new` and `new[]` can use completely different memory allocation schemes.

```
Simple* mySimpleArray = new Simple[4];
// Use mySimpleArray...
delete [] mySimpleArray;
mySimpleArray = nullptr;
```

WARNING *Always use `delete` on anything allocated with `new`, and always use `delete[]` on anything allocated with `new[]`.*

Of course, the destructors are called only if the elements of the array are objects. If you have an array of pointers, you will still need to delete each object pointed to individually just as you allocated each object individually, as shown in the following code:

```
size_t arrSize = 4;
Simple** mySimplePtrArray = new Simple*[arrSize];
// Allocate an object for each pointer.
for (size_t i = 0; i < arrSize; i++) {
    mySimplePtrArray[i] = new Simple();
}
// Use mySimplePtrArray...
// Delete each allocated object.
for (size_t i = 0; i < arrSize; i++) {
    delete mySimplePtrArray[i];
}
// Delete the array itself.
delete [] mySimplePtrArray;
mySimplePtrArray = nullptr;
```

WARNING *In modern C++ you should avoid using naked C-style pointers. Thus, instead of storing plain-old dumb pointers in plain-old dumb arrays, you should store smart pointers in modern containers. These smart pointers will automatically deallocate memory associated with them.*

Multi-Dimensional Arrays

Multi-dimensional arrays extend the notion of indexed values to use multiple indices. For example, a Tic-Tac-Toe game might use a two-dimensional array to represent a three-by-three grid. The following example shows such an array declared on the stack and accessed with some test code:

```
char board[3][3];
// Test code
board[0][0] = 'X';    // X puts marker in position (0,0).
board[2][1] = 'O';    // O puts marker in position (2,1).
```

You may be wondering whether the first subscript in a two-dimensional array is the x-coordinate or the y-coordinate. The truth is that it doesn't really matter, as long as you are consistent. A four-by-seven grid could be declared as `char board[4][7]` or `char board[7][4]`. For most applications, it is easiest to think of the first subscript as the x-axis and the second as the y-axis.

Multi-Dimensional Stack Arrays

In memory, a stack-based two-dimensional array looks like Figure 22-8. Because memory doesn't have two axes (addresses are merely sequential), the computer represents a two-dimensional array just like a one-dimensional array. The difference is the size of the array and the method used to access it.

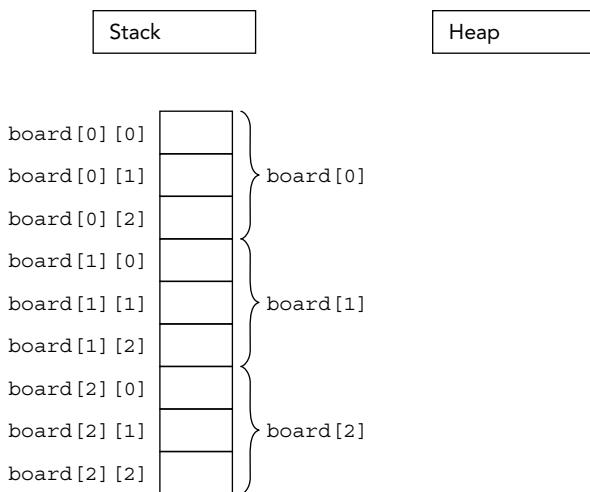


FIGURE 22-8

The size of a multi-dimensional array is all of its dimensions multiplied together, then multiplied by the size of a single element in the array. In Figure 22-8, the three-by-three board is $3 \times 3 \times 1 = 9$ bytes, assuming that a character is 1 byte. For a four-by-seven board of characters, the array would be $4 \times 7 \times 1 = 28$ bytes.

To access a value in a multi-dimensional array, the computer treats each subscript as accessing another subarray within the multi-dimensional array. For example, in the three-by-three grid, the expression `board[0]` actually refers to the subarray highlighted in Figure 22-9. When you add a second subscript, such as `board[0] [2]`, the computer is able to access the correct element by looking up the second subscript within the subarray, as shown in Figure 22-10.

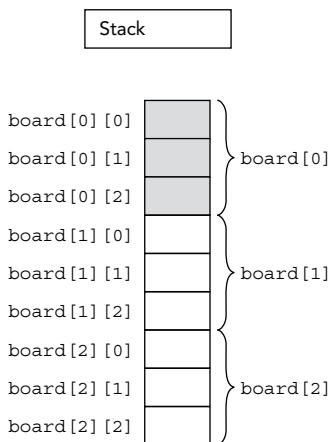


FIGURE 22-9

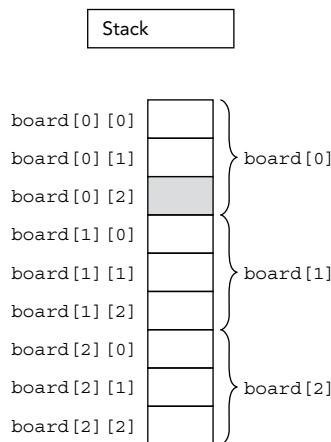


FIGURE 22-10

These techniques are extended to N -dimensional arrays, though dimensions higher than three tend to be difficult to conceptualize and are rarely useful in everyday applications.

Multi-Dimensional Heap Arrays

If you need to determine the dimensions of a multi-dimensional array at run time, you can use a heap-based array. Just as a single-dimensional dynamically allocated array is accessed through a pointer, a multi-dimensional dynamically allocated array is also accessed through a pointer. The only difference is that in a two-dimensional array, you need to start with a pointer-to-a-pointer; and in an N -dimensional array, you need N levels of pointers. At first, it might seem as if the correct way to declare and allocate a dynamically allocated multi-dimensional array is as follows:

```
char** board = new char[i][j]; // BUG! Doesn't compile
```

This code doesn't compile because heap-based arrays don't work like stack-based arrays. Their memory layout isn't contiguous, so allocating enough memory for a stack-based multi-dimensional array is incorrect. Instead, you can start by allocating a single contiguous array for the first subscript dimension of a heap-based array. Each element of that array is actually a pointer to another array that stores the elements for the second subscript dimension. This layout for a two-by-two dynamically allocated board is shown in Figure 22-11.

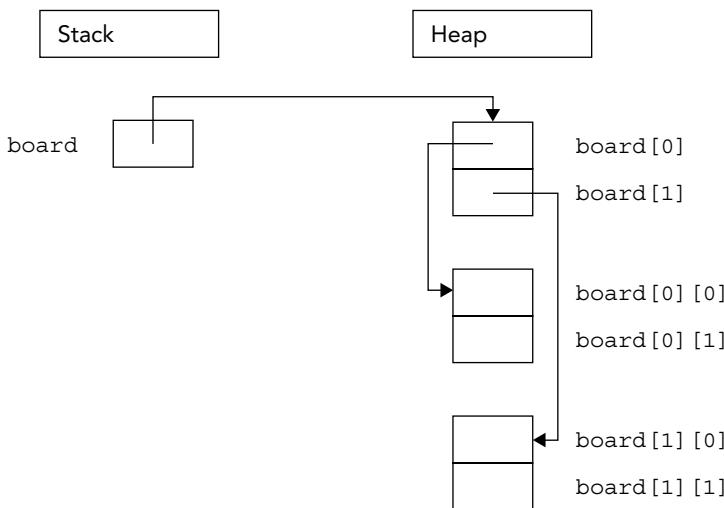


FIGURE 22-11

Unfortunately, the compiler doesn't allocate memory for the subarrays on your behalf. You can allocate the first dimension array just like a single-dimensional heap-based array, but the individual subarrays must be explicitly allocated. The following function properly allocates memory for a two-dimensional array:

```
char** allocateCharacterBoard(size_t xDimension, size_t yDimension)
{
    char** myArray = new char*[xDimension]; // Allocate first dimension
    for (size_t i = 0; i < xDimension; i++) {
```

```

        myArray[i] = new char[yDimension]; // Allocate ith subarray
    }
    return myArray;
}

```

When you wish to release the memory associated with a multi-dimensional heap-based array, the array `delete[]` syntax will not clean up the subarrays on your behalf. Your code to release an array should mirror the code to allocate it, as in the following function:

```

void releaseCharacterBoard(char** myArray, size_t xDimension)
{
    for (size_t i = 0; i < xDimension; i++) {
        delete [] myArray[i]; // Delete ith subarray
    }
    delete [] myArray; // Delete first dimension
}

```

WARNING Now that you know all the details to work with arrays, it is recommended to avoid these old C-style arrays as much as possible because they do not provide any memory safety. They are explained here because you will encounter them in legacy code. In new code, you should use the C++ STL containers such as `std::array`, `std::vector`, `std::list`, and so on. For example, use `vector<T>` for a one-dimensional dynamic array. Use `vector<vector<T>>` for a two-dimensional dynamic array and so on.

Working with Pointers

Pointers get their bad reputation from the relative ease with which you can abuse them. Because a pointer is just a memory address, you could theoretically change that address manually, even doing something as scary as the following line of code:

```
char* scaryPointer = (char*)7;
```

The previous line builds a pointer to the memory address 7, which is likely to be random garbage or memory used elsewhere in the application. If you start to use areas of memory that weren't set aside on your behalf with `new`, eventually you will corrupt the memory associated with an object, or the memory involved with the management of the heap, and your program will malfunction. Such a malfunction can manifest itself in several ways. For example, it can manifest itself as invalid results because the data has been corrupted, or as hardware exceptions being triggered due to accessing non-existent memory, or attempting to write to protected memory. If you are lucky, you will get one of the serious errors that usually results in program termination by the operating system or the C++ runtime library; if you are unlucky, you will just get a wrong result.

A Mental Model for Pointers

There are two ways to think about pointers. More mathematically minded readers might view pointers as addresses. This view makes pointer arithmetic, covered later in this chapter, a bit easier to understand. Pointers aren't mysterious pathways through memory; they are numbers that happen

to correspond to a location in memory. Figure 22-12 illustrates a two-by-two grid in the address-based view of the world.

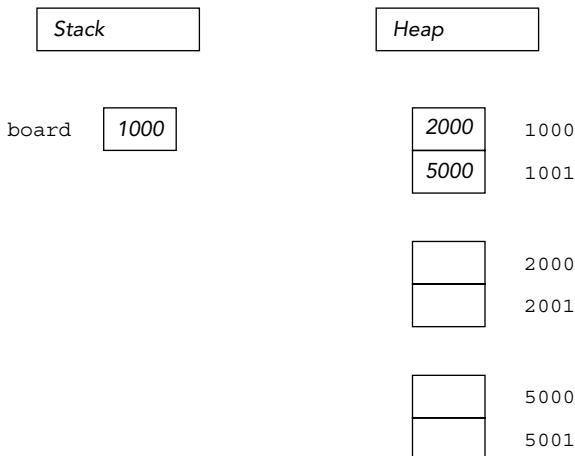


FIGURE 22-12

NOTE *The addresses in Figure 22-12 are just for illustrative purposes. Addresses on a real system are highly dependent on your hardware and operating system.*

Readers who are more comfortable with spatial representations might derive more benefit from the “arrow” view of pointers. A pointer is a level of indirection that says to the program, “Hey! Look over there.” With this view, multiple levels of pointers become individual steps on the path to data. Figure 22-11 shows a graphical view of pointers in memory.

When you *dereference* a pointer, by using the `*` operator, you are telling the program to look one level deeper in memory. In the address-based view, think of a dereference as a jump in memory to the address indicated by the pointer. With the graphical view, every dereference corresponds to following an arrow from its base to its head.

When you take the address of a location, using the `&` operator, you are adding a level of indirection in memory. In the address-based view, the program is noting the numerical address of the location, which can be stored as a pointer. In the graphical view, the `&` operator creates a new arrow whose head ends at the location designated by the expression. The base of the arrow can be stored as a pointer.

Casting with Pointers

Because pointers are just memory addresses (or arrows to somewhere), they are somewhat weakly typed. A pointer to an XML Document is the same size as a pointer to an integer. The compiler will let you easily cast any pointer type to any other pointer type using a C-style cast:

```
Document* documentPtr = getDocument();
char* myCharPtr = (char*)documentPtr;
```

A static cast offers a bit more safety. The compiler will refuse to perform a static cast on pointers to different data types:

```
Document* documentPtr = getDocument();
char* myCharPtr = static_cast<char*>(documentPtr); // BUG! Won't compile
```

If the two pointers you are casting are actually pointing to objects that are related through inheritance, the compiler will permit a static cast. However, a dynamic cast is a safer way to accomplish a cast within an inheritance hierarchy. Consult Chapter 10 for details on all C++ style casts.

ARRAY-POINTER DUALITY

You have already seen some of the overlap between pointers and arrays. Heap-allocated arrays are referred to by a pointer to their first element. Stack-based arrays are referred to by using the array syntax ([]) with an otherwise normal variable declaration. As you are about to learn, however, the overlap doesn't end there. Pointers and arrays have a complicated relationship.

Arrays Are Pointers!

A heap-based array is not the only place where you can use a pointer to refer to an array. You can also use the pointer syntax to access elements of a stack-based array. The address of an array is really the address of the 0th element. The compiler knows that when you refer to an array in its entirety by its variable name, you are really referring to the address of the 0th element. In this way, the pointer works just like a heap-based array. The following code creates an array on the stack, but uses a pointer to access the array:

```
int myIntArray[10];
int* myIntPtr = myIntArray;
// Access the array through the pointer.
myIntPtr[4] = 5;
```

The ability to refer to a stack-based array through a pointer is useful when passing arrays into functions. The following function accepts an array of integers as a pointer. Note that the caller will need to explicitly pass in the size of the array because the pointer implies nothing about size. In fact, C++ arrays of any form, pointer or not, have no built-in notion of size. That is another reason why you should use modern containers such as those provided by the STL.

```
void doubleInts(int* theArray, size_t inSize)
{
    for (size_t i = 0; i < inSize; i++) {
        theArray[i] *= 2;
    }
}
```

The caller of this function can pass a stack-based or heap-based array. In the case of a heap-based array, the pointer already exists and is passed by value into the function. In the case of a stack-based array, the caller can pass the array variable, and the compiler will automatically treat the array

variable as a pointer to the array, or you can explicitly pass the address of the first element. All three forms are shown here:

```
size_t arrSize = 4;
int* heapArray = new int[arrSize] { 1, 5, 3, 4 };
doubleInts(heapArray, arrSize);
delete [] heapArray;
heapArray = nullptr;

int stackArray[] = { 5, 7, 9, 11 };
arrSize = sizeof(stackArray) / sizeof(stackArray[0]);
doubleInts(stackArray, arrSize);
doubleInts(&stackArray[0], arrSize);
```

Parameter-passing semantics of arrays are uncannily similar to that of pointers, because the compiler treats an array as a pointer when it is passed to a function. A function that takes an array as an argument and changes values inside the array is actually changing the original array, not a copy. Just like a pointer, passing an array effectively mimics pass-by-reference functionality because what you really pass to the function is the address of the original array, not a copy. The following implementation of `doubleInts()` changes the original array even though the parameter is an array, not a pointer:

```
void doubleInts(int theArray[], size_t inSize)
{
    for (size_t i = 0; i < inSize; i++) {
        theArray[i] *= 2;
    }
}
```

Any number between the square brackets after `theArray` in the function prototype is simply ignored. The following three versions are identical:

```
void doubleInts(int* theArray, size_t inSize);
void doubleInts(int theArray[], size_t inSize);
void doubleInts(int theArray[2], size_t inSize);
```

You may be wondering why things work this way. Why doesn't the compiler just copy the array when array syntax is used in the function definition? This is done for efficiency — it takes time to copy the elements of an array, and they potentially take up a lot of memory. By always passing a pointer, the compiler doesn't need to include the code to copy the array.

There is a way to pass known-length stack-based arrays “by reference” to a function, although the syntax is non-obvious. This does not work for heap-based arrays. For example, the following `doubleIntsStack()` accepts only stack-based arrays of size 4:

```
void doubleIntsStack(int (&theArray)[4]);
```

A function template can be used to let the compiler deduce the size of the stack-based array automatically:

```
template<size_t N>
void doubleIntsStack(int (&theArray)[N])
{
    for (size_t i = 0; i < N; i++) {
```

```
    theArray[i] *= 2;
}
}
```

To summarize, arrays declared using array syntax can be accessed through a pointer. When an array is passed to a function, it is always passed as a pointer.

Not All Pointers Are Arrays!

Because the compiler lets you pass in an array where a pointer is expected, as in the `doubleInts()` function shown earlier, you may be led to believe that pointers and arrays are the same. In fact there are subtle, but important, differences. Pointers and arrays share many properties and can sometimes be used interchangeably (as shown earlier), but they are not the same.

A pointer by itself is meaningless. It may point to random memory, a single object, or an array. You can always use array syntax with a pointer, but doing so is not always appropriate because pointers aren't always arrays. For example, consider the following code:

```
int* ptr = new int;
```

The pointer `ptr` is a valid pointer, but it is not an array. You can access the pointed-to value using array syntax (`ptr[0]`), but doing so is stylistically questionable and provides no real benefit. In fact, using array syntax with non-array pointers is an invitation for bugs. The memory at `ptr[1]` could be anything!

WARNING *Arrays are automatically referenced as pointers, but not all pointers are arrays.*

LOW-LEVEL MEMORY OPERATIONS

One of the great advantages of C++ over C is that you don't need to worry quite as much about memory. If you code using objects, you just need to make sure that each individual class properly manages its own memory. Through construction and destruction, the compiler helps you manage memory by telling you when to do it. Hiding the management of memory within classes makes a huge difference in usability, as demonstrated by the STL classes. However, with some applications or with legacy code, you may encounter the need to work with memory at a lower level. Whether for legacy, efficiency, debugging, or curiosity, knowing some techniques for working with raw bytes can be helpful.

Pointer Arithmetic

The C++ compiler uses the declared types of pointers to allow you to perform *pointer arithmetic*. If you declare a pointer to an `int` and increase it by 1, the pointer moves ahead in memory by the size of an `int`, not by a single byte. This type of operation is most useful with arrays, since they contain

homogeneous data that is sequential in memory. For example, assume you declare an array of ints on the heap:

```
int* myArray = new int[8];
```

You are already familiar with the following syntax for setting the value in position 2:

```
myArray[2] = 33;
```

With pointer arithmetic, you can equivalently use the following syntax, which obtains a pointer to the memory that is “2 ints ahead” of `myArray` and then dereferences it to set the value:

```
* (myArray + 2) = 33;
```

As an alternative syntax for accessing individual elements, pointer arithmetic doesn’t seem too appealing. Its real power lies in the fact that an expression like `myArray + 2` is still a pointer to an `int`, and thus can represent a smaller `int` array. Suppose you have the following wide string:

```
const wchar_t* myString = L"Hello, World!";
```

Suppose you also have a function that takes in a string and returns a new string that contains a capitalized version of the input:

```
wchar_t* toCaps(const wchar_t* inString);
```

You can capitalize `myString` by passing it into this function. However, if you only want to capitalize *part* of `myString`, you can use pointer arithmetic to refer to only a latter part of the string. The following code calls `toCaps()` on the `World` part of the string by just adding 7 to the pointer, even though `wchar_t` is usually more than 1 byte:

```
toCaps(myString + 7);
```

Another useful application of pointer arithmetic involves subtraction. Subtracting one pointer from another of the same type gives you the number of elements of the pointed-to type between the two pointers, not the absolute number of bytes between them.

Custom Memory Management

For 99 percent of the cases you will encounter (some might say 100 percent of the cases), the built-in memory allocation facilities in C++ are adequate. Behind the scenes, `new` and `delete` do all the work of handing out memory in properly sized chunks, maintaining a list of available areas of memory, and releasing chunks of memory back to that list upon deletion.

When resource constraints are extremely tight, or under very special conditions, such as managing shared memory, implementing custom memory management may be a viable option. Don’t worry — it’s not as scary as it sounds. Basically, managing memory yourself generally means that classes allocate a large chunk of memory and dole out that memory in pieces as it is needed.

How is this approach any better? Managing your own memory can potentially reduce overhead. When you use `new` to allocate memory, the program also needs to set aside a small amount of space to record how much memory was allocated. That way, when you call `delete`, the proper amount of memory can be released. For most objects, the overhead is so much smaller than the memory

allocated that it makes little difference. However, for small objects or programs with enormous numbers of objects, the overhead can have an impact.

When you manage memory yourself, you might know the size of each object a priori, so you might be able to avoid the overhead for each object. The difference can be enormous for large numbers of small objects. The syntax for performing custom memory management is described in Chapter 14.

Garbage Collection

At the other end of the memory hygiene spectrum lies *garbage collection*. With environments that support garbage collection, the programmer rarely, if ever, explicitly frees memory associated with an object. Instead, objects to which there are no references anymore will be cleaned up automatically at some point by the runtime library.

Garbage collection is not built into the C++ language as it is in C# and Java. In modern C++ you use smart pointers to manage memory, while in legacy code you will see memory management at the object level through `new` and `delete`. It is possible but not easy to implement garbage collection in C++, but freeing yourself from the task of releasing memory would probably introduce new headaches.

One approach to garbage collection is called *mark and sweep*. With this approach, the garbage collector periodically examines every single pointer in your program and annotates the fact that the referenced memory is still in use. At the end of the cycle, any memory that hasn't been marked is deemed to be not in-use and is freed.

A mark-and-sweep algorithm could be implemented in C++ if you were willing to do the following:

1. Register all pointers with the garbage collector so that it can easily walk through the list of all pointers.
2. Derive all objects from a mixin class, perhaps `GarbageCollectible`, that allows the garbage collector to mark an object as in-use.
3. Protect concurrent access to objects by making sure that no changes to pointers can occur while the garbage collector is running.

As you can see, this approach to garbage collection requires quite a bit of diligence on the part of the programmer. It may even be more error-prone than using `delete!` Attempts at a safe and easy mechanism for garbage collection have been made in C++, but even if a perfect implementation of garbage collection in C++ came along, it wouldn't necessarily be appropriate to use for all applications. Among the downsides of garbage collection are the following:

- When the garbage collector is actively running, the program might become unresponsive.
- With garbage collectors, you have so called non-deterministic destructors. Because an object is not destroyed until it is garbage collected, the destructor is not executed immediately when the object leaves its scope. This means that cleaning up resources (such as closing a file, releasing a lock, etc.), which is done by the destructor, is not performed until some indeterminate time in the future.

Object Pools

Garbage collection is like buying plates for a picnic and leaving any used plates out in the yard where the wind will conveniently blow them into the neighbor's yard. Surely, there must be a more ecological approach to memory management.

Object pools are the analog of recycling. You buy a reasonable number of plates, and after using a plate, you clean it so that it can be reused later. Object pools are ideal for situations where you need to use many objects of the same type over time, and creating each one incurs overhead.

Chapter 25 contains further details on using object pools for performance efficiency.

Function Pointers

You don't normally think about the location of functions in memory, but each function actually lives at a particular address. In C++, you can use *functions as data*. In other words, you can take the address of a function and use it like you use a variable.

Function pointers are typed according to the parameter types and return type of compatible functions. One way to work with function pointers is to use a type alias, discussed in Chapter 10. A type alias allows you to assign a type name to the family of functions that have the given characteristics. For example, the following line defines a type called `MatchFcn` that represents a pointer to any function that has two `int` parameters and returns a `bool`:

```
using MatchFcn = bool(*)(int, int);
```

This type alias is equivalent to the following old-school, less readable, `typedef`:

```
typedef bool(*MatchFcn)(int, int);
```

Now that this new type exists, you can write a function that takes a `MatchFcn` as a parameter. For example, the following function accepts two `int` arrays and their size, as well as a `MatchFcn`. It iterates through the arrays in parallel and calls the `MatchFcn` on corresponding elements of both arrays, printing a message if the `MatchFcn` function returns `true`. Notice that even though the `MatchFcn` is passed in as a variable, it can be called just like a regular function:

```
void findMatches(int values1[], int values2[], size_t numValues, MatchFcn inFunc)
{
    for (size_t i = 0; i < numValues; i++) {
        if (inFunc(values1[i], values2[i])) {
            cout << "Match found at position " << i <<
                " (" << values1[i] << ", " << values2[i] << ")" << endl;
        }
    }
}
```

Note that this implementation requires that both arrays have at least `numValues` elements. To call the `findMatches()` function, all you need is any function that adheres to the defined `MatchFcn` type —

that is, any type that takes in two `ints` and returns a `bool`. For example, consider the following function, which returns `true` if the two parameters are equal:

```
bool intEqual(int inItem1, int inItem2)
{
    return inItem1 == inItem2;
}
```

Because the `intEqual()` function matches the `MatchFcn` type, it can be passed as the final argument to `findMatches()`, as follows:

```
int arr1[] = { 2, 5, 6, 9, 10, 1, 1 };
int arr2[] = { 4, 4, 2, 9, 0, 3, 4 };
int arrSize = sizeof(arr1) / sizeof(arr1[0]);
cout << "Calling findMatches() using intEqual(): " << endl;
findMatches(arr1, arr2, arrSize, &intEqual);
```

The `intEqual()` function is passed into the `findMatches()` function by taking its address. Technically, the `&` character is optional — if you put the function name, the compiler will know that you mean to take its address. The output is as follows:

```
Calling findMatches() using intEqual():
Match found at position 3 (9, 9)
```

The benefit of function pointers lies in the fact that `findMatches()` is a generic function that compares parallel values in two arrays. As it is used above, it compares based on equality. However, because it takes a function pointer, it could compare based on other criteria. For example, the following function also adheres to the definition of `MatchFcn`:

```
bool bothOdd(int inItem1, int inItem2)
{
    return inItem1 % 2 == 1 && inItem2 % 2 == 1;
}
```

The following code calls `findMatches()` using `bothOdd`:

```
cout << "Calling findMatches() using bothOdd(): " << endl;
findMatches(arr1, arr2, arrSize, &bothOdd);
```

The output is:

```
Calling findMatches() using bothOdd():
Match found at position 3 (9, 9)
Match found at position 5 (1, 3)
```

By using function pointers, a single function, `findMatches()`, is customized to different uses based on a parameter, `inFunc`.

NOTE Instead of using these old-style function pointers, you can also use `std::function`, which is explained in Chapter 19.

Pointers to Methods and Data Members

You can create and use pointers to both variables and functions. Now, consider pointers to class data members and methods. It's perfectly legitimate in C++ to take the addresses of class data members and methods in order to obtain pointers to them. However, you can't access a non-static data member or call a non-static method without an object. The whole point of class data members and methods is that they exist on a per-object basis. Thus, when you want to call the method or access the data member via the pointer, you must dereference the pointer in the context of an object. Here is an example:

```
SpreadsheetCell myCell(123);
double (SpreadsheetCell::*methodPtr) () const = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr) () << endl;
```

Don't panic at the syntax. The second line declares a variable called `methodPtr` of type pointer to a non-static `const` method of `SpreadsheetCell` that takes no arguments and returns a `double`. At the same time, it initializes this variable to point to the `getValue()` method of the `SpreadsheetCell` class. This syntax is quite similar to declaring a simple function pointer, except for the addition of `SpreadsheetCell::` before the `*methodPtr`. Note also that the `&` is required in this case.

The third line calls the `getValue()` method (via the `methodPtr` pointer) on the `myCell` object. Note the use of parentheses surrounding `myCell.*methodPtr`. They are needed because `()` has higher precedence than `*`.

The second line can be simplified with a type alias:

```
SpreadsheetCell myCell(123);
using PtrToGet = double (SpreadsheetCell::*) () const;
PtrToGet methodPtr = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr) () << endl;
```

Using `auto` it can be simplified even further:

```
SpreadsheetCell myCell(123);
auto methodPtr = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr) () << endl;
```

Pointers to methods and data members usually won't come up in your programs. However, it's important to keep in mind that you can't dereference a pointer to a non-static method or data member without an object. Every so often, you'll find yourself wanting to try something like passing a pointer to a non-static method to a function such as `qsort()` that requires a function pointer, which simply won't work .

NOTE C++ does permit you to dereference a pointer to a static data member or method without an object.

SMART POINTERS

Memory management in C++ is a perennial source of errors and bugs. Many of these bugs arise from the use of dynamic memory allocation and pointers. When you extensively use dynamic memory allocation in your program and pass many pointers between objects, it's difficult to

remember to call `delete` on each pointer exactly once and at the right time. The consequences of getting it wrong are severe: When you free dynamically allocated memory more than once you can cause memory corruption or a fatal run-time error, and when you forget to free dynamically allocated memory you cause memory leaks.

Smart pointers help you manage your dynamically allocated memory and are the recommended technique for avoiding memory leaks. Smart pointers are a notion that arose from the fact that most memory-related issues are avoidable by putting everything on the stack. The stack is much safer than the heap because stack variables are automatically destructed and cleaned up when they go out of scope. Smart pointers combine the safety of stack variables with the flexibility of heap variables. There are several kinds of smart pointers. The simplest type of smart pointer takes sole ownership of the memory and when the smart pointer goes out of scope, it frees the referenced memory, for example, `unique_ptr`.

However, managing pointers presents more problems than just remembering to delete them when they go out of scope. Sometimes several objects or pieces of code contain copies of the same pointer. This problem is called *aliasing*. In order to free all memory properly, the last piece of code to use the memory should call `delete` on the pointer. However, it is often difficult to know which piece of code uses the memory last. It may even be impossible to determine the order when you code because it might depend on run-time inputs. Thus, a more sophisticated type of smart pointer implements reference counting to keep track of its owners. When all owners are finished using the pointer, the number of references drops to 0 and the smart pointer calls `delete` on its underlying dumb pointer. The standard `shared_ptr` smart pointer discussed later includes reference counting. It is important to be familiar with this concept.

C++ provides several language features that make smart pointers attractive. First, you can write a type-safe smart pointer class for any pointer type using templates. Second, you can provide an interface to the smart pointer objects using operator overloading that allows code to use the smart pointer objects as if they were dumb pointers. Specifically, you can overload the `*` and `->` operators such that client code can dereference a smart pointer object the same way it dereferences a normal pointer.

The Old Deprecated `auto_ptr`

The old, pre-C++11 standard library included a basic implementation of a smart pointer, called `auto_ptr`. Unfortunately, `auto_ptr` has some serious shortcomings. One of these shortcomings is that it does not work correctly when used inside STL containers like `vectors`. C++ has now officially deprecated `auto_ptr` and replaced it with `unique_ptr` and `shared_ptr`, discussed in the next section. `auto_ptr` is mentioned here to make sure you know about it and to make sure you never use it.

WARNING *Do not use the old `auto_ptr` smart pointer anymore. Instead use `unique_ptr` or `shared_ptr`!*

The unique_ptr and shared_ptr Smart Pointers

The difference between `shared_ptr` and `unique_ptr` is that `shared_ptr` is a reference-counted smart pointer, while `unique_ptr` is not reference counted. This means that you can have several `shared_ptr` instances pointing to the same dynamically allocated memory and the memory will only be deallocated when the last `shared_ptr` goes out of scope. `shared_ptr` is also thread-safe. See Chapter 23 for a discussion on multithreading.

`unique_ptr` on the other hand means ownership. When the single `unique_ptr` goes out of scope, the underlying memory is freed.

Your default smart pointer should be `unique_ptr`. Use only `shared_ptr` when you need to share the resource.

Both smart pointers require you to include the `<memory>` header file.

WARNING *Never assign the result of a memory allocation to a dumb pointer. Whatever memory allocation method you use, always immediately give the memory pointer to a smart pointer, `unique_ptr` or `shared_ptr`.*

unique_ptr

As a rule of thumb, always store dynamically allocated objects in stack-based instances of `unique_ptr`. For example, consider the following function that blatantly leaks memory by allocating a `Simple` object on the heap and neglecting to release it:

```
void leaky()
{
    Simple* mySimplePtr = new Simple(); // BUG! Memory is never released!
    mySimplePtr->go();
}
```

Sometimes you might think that your code is properly deallocating dynamically allocated memory. Unfortunately, it most likely is *not correct* in all situations. Take the following function:

```
void couldBeLeaky()
{
    Simple* mySimplePtr = new Simple();
    mySimplePtr->go();
    delete mySimplePtr;
}
```

The above function dynamically allocates a `Simple` object, uses the object, and then properly calls `delete`. However, you can still have memory leaks in this example! If the `go()` method throws an exception, the call to `delete` is never executed, causing a memory leak.

In both cases you should use a `unique_ptr`. The object is not explicitly deleted; but when the `unique_ptr` instance goes out of scope (at the end of the function, or because an exception is thrown) it automatically deallocates the `Simple` object in its destructor:

```
void notLeaky()
{
    auto mySimpleSmartPtr = make_unique<Simple>();
    mySimpleSmartPtr->go();
}
```

This code uses `make_unique()` from C++14, in combination with the `auto` keyword, so that you only have to specify the type of the pointer, `Simple` in this case, once. If the `Simple` constructor requires parameters, you put them in between the parentheses of the `make_unique()` call.

If your compiler does not yet support `make_unique()` you can create your `unique_ptr` as follows. Note that `Simple` must be mentioned twice.

```
unique_ptr<Simple> mySimpleSmartPtr(new Simple());
```

WARNING *Always use `make_unique()` to create a `unique_ptr`, if your compiler supports it.*

One of the greatest characteristics of smart pointers is that they provide enormous benefit without requiring the user to learn a lot of new syntax. As you can see in the preceding code, the smart pointer can still be dereferenced (using `*` or `->`) just like a standard pointer.

A `unique_ptr` is suitable to store a dynamically allocated old C-style array. For example, the following example creates a `unique_ptr` that holds a dynamically allocated C-style array of 10 integers:

```
auto myVariableSizedArray = make_unique<int[]>(10);
```

Even though it is possible to use a `unique_ptr` to store a dynamically allocated C-style array, it's recommended to use an STL container instead, such as `std::array` or `std::vector`.

By default, `unique_ptr` uses the standard `new` and `delete` operators to allocate and deallocate memory. You can change this behavior as follows:

```
int* malloc_int(int value)
{
    int* p = (int*)malloc(sizeof(int));
    *p = value;
    return p;
}
int main()
{
    auto deleter = [] (int* p){ free(p); };
    unique_ptr<int, decltype(deleter)> myIntSmartPtr(malloc_int(42), deleter);
    return 0;
}
```

This code allocates the memory for the integer with `malloc_int()`. The `unique_ptr` deallocates the memory by calling the named lambda, `deleter`, which uses the standard `free()` function. As said before, in C++ you should never use `malloc()` but `new` instead. However, this feature of `unique_ptr` is available because it is useful to manage other resources instead of just memory. For example, it

can be used to automatically close a file or network socket or anything when the `unique_ptr` goes out of scope.

Unfortunately, the syntax for a custom deleter with `unique_ptr` is a bit clumsy. Using a custom deleter with `shared_ptr` is much easier. The following section on `shared_ptr` demonstrates how to use a `shared_ptr` to automatically close a file when it goes out of scope.

shared_ptr

`shared_ptr` is used in a similar way as `unique_ptr`. To create one, you use `make_shared()`, which is more efficient than creating a `shared_ptr` directly. For example:

```
auto mySimpleSmartPtr = make_shared<Simple>();
```

WARNING Always use `make_shared()` to create a `shared_ptr`.

A difference with `unique_ptr` is that a `shared_ptr` cannot be used to store a pointer to a dynamically allocated old C-style array.

WARNING Never use a `shared_ptr` to manage a pointer to a C-style array. Use `unique_ptr` to manage the C-style array, or better yet, use STL containers instead of C-style arrays.

The following functions are available to cast `shared_ptr`s: `const_pointer_cast()`, `dynamic_pointer_cast()`, and `static_pointer_cast()`. For example, suppose you have a `Base` class, and a `Derived` class that derives from `Base`. You can then write:

```
shared_ptr<Base> myBasePtr(new Derived);
shared_ptr<Derived> myDerivedPtr = dynamic_pointer_cast<Derived>(myBasePtr);
```

Just as `unique_ptr`, `shared_ptr` by default uses the standard `new` and `delete` operators to allocate and deallocate memory. You can change this behavior as follows:

```
// Implementation of malloc_int() as before.
int main()
{
    shared_ptr<int> myIntSmartPtr(malloc_int(42), free);
    return 0;
}
```

As you can see, this is much easier than a custom deleter with `unique_ptr`.

The following example uses a `shared_ptr` to store a file pointer. When the `shared_ptr` goes out of scope, the file pointer is automatically closed with a call to the `CloseFile()` function. Remember that C++ has proper object-oriented classes to work with files (see Chapter 12). Those classes already automatically close their files when they go out of scope. This example using the old C `fopen()` and `fclose()` functions is just to give a demonstration for what `shared_ptr`s can be used for besides pure memory:

```
void CloseFile(FILE* filePtr)
{
    if (filePtr == nullptr)
        return;
    fclose(filePtr);
    cout << "File closed." << endl;
}
int main()
{
    FILE* f = fopen("data.txt", "w");
    shared_ptr<FILE> filePtr(f, CloseFile);
    if (filePtr == nullptr) {
        cerr << "Error opening file." << endl;
    } else {
        cout << "File opened." << endl;
        // Use filePtr
    }
    return 0;
}
```

The Need for Reference Counting

As a general concept, *reference counting* is the technique for keeping track of the number of instances of a class or particular object in use. A reference-counting smart pointer is one that keeps track of how many smart pointers have been built to refer to a single real pointer, or single object. This way, smart pointers can avoid double deletion.

The double deletion problem is easy to provoke. Consider the following class, `Nothing`, which prints out messages when an object is created and destroyed:

```
class Nothing
{
public:
    Nothing() { cout << "Nothing::Nothing()" << endl; }
    virtual ~Nothing() { cout << "Nothing::~Nothing()" << endl; }
};
```

If you were to create two standard `shared_ptr`s and have them both refer to the same `Nothing` object as follows, both smart pointers would attempt to delete the same object when they go out of scope:

```
void doubleDelete()
{
    Nothing* myNothing = new Nothing();
    shared_ptr<Nothing> smartPtr1(myNothing);
    shared_ptr<Nothing> smartPtr2(myNothing);
}
```

Depending on your compiler, this piece of code might crash! If you do get output, the output can be:

```
Nothing::Nothing()
Nothing::~Nothing()
Nothing::~Nothing()
```

Yikes! One call to the constructor and two calls to the destructor? You get the same problem with `unique_ptr` and with the old deprecated `auto_ptr`. You might be surprised that even the

reference-counted `shared_ptr` class behaves this way. However, this is correct behavior according to the C++ standard. You should not use `shared_ptr` like in the previous `doubleDelete()` function to create two `shared_ptr`s pointing to the same object. Instead, you should use `make_shared()`, and use the copy constructor to make a copy as follows:

```
void noDoubleDelete()
{
    auto smartPtr1 = make_shared<Nothing>();
    shared_ptr<Nothing> smartPtr2(smartPtr1);
}
```

The output of this code is:

```
Nothing::Nothing()
Nothing::~Nothing()
```

Even though there are two `shared_ptr`s pointing to the same `Nothing` object, the `Nothing` object is destroyed only once. Remember that `unique_ptr` is not reference counted. In fact, `unique_ptr` will not allow you to use the copy constructor as in the `noDoubleDelete()` function.

NOTE *If your program uses smart pointers by copying them, assigning them, or passing them by value as arguments to functions, the `shared_ptr` is the perfect solution.*

If you really need to write code as shown in the previous `doubleDelete()` function, you will need to implement your own smart pointer to prevent double deletion. But again, it is recommended to use the standard `shared_ptr` template for sharing a resource. Simply avoid code as in the `doubleDelete()` function, and use the copy constructor instead:

```
shared_ptr<Nothing> smartPtr2(smartPtr1);
```

Move Semantics

Both `shared_ptr` and `unique_ptr` support move semantics discussed in Chapter 10 to make them efficient. Because of this, it is also efficient to return a `shared_ptr` or a `unique_ptr` from a function. For example, you can write the following `func()` function and use it as demonstrated in `main()`:

```
// ... definition of Simple not shown for brevity
shared_ptr<Simple> func()
{
    auto ptr = make_shared<Simple>();
    // Do something with ptr...
    return ptr;
}
int main()
{
    shared_ptr<Simple> mySmartPtr = func();
    return 0;
}
```

This is efficient because C++ automatically calls `std::move()` on the return statement in `func()`, which triggers the move semantics of `shared_ptr`. The same happens if you use `unique_ptr` instead of `shared_ptr`. `unique_ptr` does not support the normal copy assignment operator and copy constructor, but it does support the move assignment operator and move constructor, which explains why you can return a `unique_ptr` from a function.

Take a look at the following `unique_ptr` statements:

```
auto p1 = make_unique<int>(42);
unique_ptr<int> p2 = p1;           // Error: does not compile
unique_ptr<int> p3 = move(p1);    // OK
```

The line defining `p2` will not compile because it is trying to use the copy constructor, which is not available for `unique_ptr`. The definition of `p3` is good because `std::move()` (defined in the `<utility>` header file, see Chapter 10) is used to trigger the move constructor of `unique_ptr`. After `p3` is created, `p1` is reset to `nullptr` and you will be able to access the integer 42 only through `p3`; ownership has been moved from `p1` to `p3`.

weak_ptr

There is one more class in C++ that is related to `shared_ptr`; called `weak_ptr`. A `weak_ptr` can contain a reference to memory managed by a `shared_ptr`. The `weak_ptr` does not own the memory, so the `shared_ptr` is not prevented from deallocating the memory. A `weak_ptr` does not destroy the pointed to memory when it goes out of scope; however, it can be used to determine if the memory has been deallocated by the associated `shared_ptr` or not. The constructor of a `weak_ptr` requires a `shared_ptr` or another `weak_ptr` as argument. To get access to the pointer stored in a `weak_ptr` you need to convert it to a `shared_ptr`. There are two ways to do this:

- Use the `lock()` method on a `weak_ptr` instance, which returns a `shared_ptr`, or
- Create a new `shared_ptr` instance and give a `weak_ptr` as argument to the `shared_ptr` constructor.

In both cases, this new `shared_ptr` will be `nullptr` if the `shared_ptr` associated with the `weak_ptr` has been deallocated in the meantime.

COMMON MEMORY PITFALLS

It is difficult to pinpoint the exact situations that can lead to a memory-related bug. Every memory leak or bad pointer has its own nuances. There is no magic bullet for resolving memory issues, but there are several common categories of problems and some tools you can use to detect and resolve them.

Underallocating Strings

The most common problem with C-style strings is *underallocation*. In most cases, this arises when the programmer fails to allocate an extra character for the trailing '`\0`' sentinel. Underallocation of strings also occurs when programmers assume a certain fixed maximum size. The basic built-in

C-style string functions do not adhere to a fixed size — they will happily write off the end of the string into uncharted memory.

The following code demonstrates underallocation. It reads data off a network connection and puts it in a C-style string. This is done in a loop because the network connection receives only a small amount of data at a time. On each loop, `getMoreData()` is called, which returns a pointer to dynamically allocated memory. When `nullptr` is returned from `getMoreData()`, all of the data has been received. `strcat()` is a C function which concatenates the C-style string given as a second argument to the end of the C-style string given as a first argument. It expects the destination buffer to be big enough.

```
char buffer[1024] = {0}; // Allocate a whole bunch of memory.
while (true) {
    char* nextChunk = getMoreData();
    if (nextChunk == nullptr) {
        break;
    } else {
        strcat(buffer, nextChunk); // BUG! No guarantees against buffer overrun!
        delete [] nextChunk;
    }
}
```

There are three ways to resolve the possible underallocation problem. In decreasing order of preference, they are:

1. Use C++-style strings, which handle the memory associated with concatenation on your behalf.
2. Instead of allocating a buffer as a global variable or on the stack, allocate it on the heap. When there is insufficient space left, allocate a new buffer large enough to hold at least the current contents plus the new chunk, copy the original buffer into the new buffer, append the new contents, and delete the original buffer.
3. Create a version of `getMoreData()` that takes a maximum count (including the '\0' character) and returns no more characters than that; then track the amount of space left and the current position in the buffer.

Accessing Out-of-Bounds Memory

Earlier in this chapter, you read that since a pointer is just a memory address, it is possible to have a pointer that points to a random location in memory. Such a condition is quite easy to fall into. For example, consider a C-style string that has somehow lost its '\0' termination character. The following function, which fills the string with all 'm' characters, continues to fill the contents of memory following the string with 'm's:

```
void fillWithM(char* inStr)
{
    int i = 0;
    while (inStr[i] != '\0') {
        inStr[i] = 'm';
```

```

        i++;
    }
}

```

If an improperly terminated string is handed to this function, it is only a matter of time before an essential part of memory is overwritten and the program crashes. Consider what might happen if the memory associated with the objects in your program is suddenly overwritten with 'm's. It's not pretty!

Bugs that result in writing to memory past the end of an array are often called *buffer overflow errors*. Such bugs have been exploited by several high-profile malware programs; for example, viruses and worms. A devious hacker can take advantage of the ability to overwrite portions of memory to inject code into a running program.

Many memory-checking tools detect buffer overflows. Also, using higher-level constructs like C++ strings and vectors help prevent numerous bugs associated with writing to C-style strings and arrays.

WARNING *Avoid using old C-style strings and arrays that offer no protection whatsoever. Instead, use modern and safe constructs like C++ strings and vectors that manage all their memory for you.*

Memory Leaks

Finding and fixing memory leaks can be one of the more frustrating parts of programming in C or C++. Your program finally works and appears to give the correct results. Then, you start to notice that your program gobbles up more and more memory as it runs. Your program has a memory leak. The use of smart pointers to avoid memory leaks is a good first approach to solving the problem.

Memory leaks occur when you allocate memory and neglect to release it. At first, this sounds like the result of careless programming that could easily be avoided. After all, if every new has a corresponding delete in every class you write, there should be no memory leaks, right? Actually, that's not always true. In the following code, the `Simple` class is properly written to release any memory that it allocates. Keep in mind that this `Simple` class implementation is only for demonstration purposes. In real code, you should make the `mIntPtr` a `unique_ptr`.

When `doSomething()` is called, the `outSimplePtr` pointer is changed to another `Simple` object without deleting the old one. The `doSomething()` function doesn't delete the old object on purpose to demonstrate memory leaks. Once you lose a pointer to an object, it's nearly impossible to delete it.

```

class Simple
{
public:
    Simple() { mIntPtr = new int(); }
    virtual ~Simple() { delete mIntPtr; }
    void setIntPtr(int inInt) { *mIntPtr = inInt; }
}

```

```

private:
    int* mIntPtr;
};

void doSomething(Simple*& outSimplePtr)
{
    outSimplePtr = new Simple(); // BUG! Doesn't delete the original.
}

int main()
{
    Simple* simplePtr = new Simple(); // Allocate a Simple object.
    doSomething(simplePtr);
    delete simplePtr; // Only cleans up the second object.
    return 0;
}

```

In cases like the preceding example, the memory leak probably arose from poor communication between programmers or poor documentation of code. The caller of `doSomething()` may not have realized that the variable was passed by reference and thus had no reason to expect that the pointer would be reassigned. If they did notice that the parameter is a non-const reference to a pointer, they may have suspected that something strange was happening, but there is no comment around `doSomething()` that explains this behavior.

Finding and Fixing Memory Leaks in Windows with Visual C++

Memory leaks are hard to track down because you can't easily look at memory and see what objects are not in use and where they were originally allocated. However, there are programs that can do this for you. Memory leak detection tools range from expensive professional software packages to free downloadable tools. If you already work with Microsoft Visual C++, its debug library has built-in support for memory leak detection. This memory leak detection is not enabled by default, unless you create an MFC project. To enable it in other projects, you need to start with including the following three lines at the beginning of your code:

```

#define _CRTDBG_MAP_ALLOC
#include <cstdlib>
#include <crtdbg.h>

```

These lines should be in the exact order as shown. Next, you need to redefine the `new` operator as follows:

```

#ifndef _DEBUG
    #ifndef DBG_NEW
        #define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
        #define new DBG_NEW
    #endif
#endif // _DEBUG

```

Note that this is within a “`#ifdef _DEBUG`” statement so the redefinition of `new` is done only when compiling a debug version of your application. This is what you normally want. Release builds usually do not do any memory leak detection.

The last thing you need to do is to add the following line as the first line in your `main()` function:

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

This tells the Visual C++ CRT (C RunTime) library to write all detected memory leaks to the debug output console when the application exits. For the previous leaky program, the debug console will contain lines similar to the following:

```
Detected memory leaks!
Dumping objects ->
c:\leaky\leaky.cpp(15) : {147} normal block at 0x014FABF8, 4 bytes long.
  Data: < 00 00 00 00
c:\leaky\leaky.cpp(33) : {146} normal block at 0x014F5048, 8 bytes long.
  Data: < 9C CC 97 00 F8 AB 4F 01
Object dump complete.
```

The output clearly shows in which file and on which line memory was allocated but never deallocated. The line number is between parentheses immediately behind the filename. The number between the curly brackets is a counter for the memory allocations. For example, {147} means the 147th allocation in your program since it started. You can use the VC++ `_CrtSetBreakAlloc()` function to tell the VC++ debug runtime to break into the debugger when a certain allocation is performed. For example, add the following line to the beginning of your `main()` function to let the debugger break on the 147th allocation:

```
_CrtSetBreakAlloc(147);
```

In this leaky program, there are two leaks — the first `Simple` object that is never deleted and the heap-based integer that it creates. In the Visual C++ debugger output window, you can simply double click on one of the memory leaks and it will automatically jump to that line in your code.

Of course, programs like Microsoft Visual C++ discussed in this section, and Valgrind discussed in the next section, can't actually fix the leak for you — what fun would that be? These tools provide information that you can use to find the actual problem. Normally, that involves stepping through the code to find out where the pointer to an object was overwritten without the original object being released. Most debuggers provide “watch point” functionality that can break execution of the program when this occurs.

Finding and Fixing Memory Leaks in Linux with Valgrind

Valgrind is an example of a free open-source tool for Linux that, amongst other things, pinpoints the exact line in your code where a leaked object was allocated.

The following output, generated by running Valgrind on the previous leaky program, pinpoints the exact locations where memory was allocated but never released. Valgrind finds the same two memory leaks — the first `Simple` object never deleted and the heap-based integer that it creates:

```
==15606== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==15606== malloc/free: in use at exit: 8 bytes in 2 blocks.
==15606== malloc/free: 4 allocs, 2 frees, 16 bytes allocated.
==15606== For counts of detected errors, rerun with: -v
==15606== searching for pointers to 2 not-freed blocks.
==15606== checked 4455600 bytes.
==15606==
```

```

==15606== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==15606==    at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==    by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606==    by 0x804875B: Simple::Simple() (leaky.cpp:4)
==15606==    by 0x8048648: main (leaky.cpp:24)
==15606==
==15606== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==15606==    at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==    by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606==    by 0x8048633: main (leaky.cpp:20)
==15606==    by 0x4031FA46: __libc_start_main (in /lib/libc-2.3.2.so)
==15606==
==15606== LEAK SUMMARY:
==15606==    definitely lost: 4 bytes in 1 blocks.
==15606==    possibly lost: 0 bytes in 0 blocks.
==15606==    still reachable: 4 bytes in 1 blocks.
==15606==          suppressed: 0 bytes in 0 blocks.

```

WARNING *It is strongly recommended to use smart pointers as much as possible to avoid memory leaks.*

Double-Deleting and Invalid Pointers

Once you release memory associated with a pointer using `delete`, the memory is available for use by other parts of your program. Nothing stops you, however, from attempting to continue to use the pointer, which is now a *dangling pointer*. Double deletion is also a problem. If you use `delete` a second time on a pointer, the program could be releasing memory that has since been assigned to another object.

Double deletion and use of already released memory are both hard problems to track down because the symptoms may not show up immediately. If two deletions occur within a relatively short amount of time, the program might work indefinitely because the associated memory is not reused that quickly. Similarly, if a deleted object is used immediately after being deleted, most likely it will still be intact.

Of course, there is no guarantee that such behavior will work or continue to work. The memory allocator is under no obligation to preserve any object once it has been deleted. Even if it does work, it is extremely poor programming style to use objects that have been deleted.

Many memory leak-checking programs, such as Microsoft Visual C++ and Valgrind, will also detect double deletion and use of released objects.

If you disregard the recommendation for using smart pointers and instead still use dumb pointers, at least set your pointers to `nullptr` after deallocating their memory. This prevents you from accidentally deleting the same pointer twice or to use an invalid pointer. It's worth noting that you are allowed to call `delete` on a `nullptr` pointer; it simply will not do anything.

SUMMARY

In this chapter, you learned the ins and outs of dynamic memory. Aside from memory-checking tools and careful coding, there are two key takeaways to avoid dynamic memory-related problems. First, you need to understand how pointers work under the hood. After reading about two different mental models for pointers, hopefully you are now confident that you know how the compiler doles out memory. Second, you can avoid all sorts of dynamic memory issues by obscuring pointers with stack-based objects, like the C++ `string` class, `vector` class, smart pointers, and so on.

If there is one thing that you should have learned from this chapter it is that you should try to avoid using old C-style constructs and functions as much as possible, and use the safe C++ alternatives.

23

Multithreaded Programming with C++

WHAT'S IN THIS CHAPTER?

- What multithreaded programming is and how to write multithreaded code
- What deadlocks and race conditions are, and how to use mutual exclusion to prevent them
- How to use atomic types and atomic operations
- Explaining thread pools

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Multithreaded programming is important on computer systems with multiple processor units. It allows you to write a program to use all those processor units in parallel. There are multiple ways for a system to have multiple processor units. The system can have multiple discrete processor chips, each one an independent CPU (Central Processor Unit). Or, the system can have a single discrete processor chip that internally consists of multiple independent CPUs, also called cores. These kind of processors are called multicore processors. A system can also have a combination of both. Systems with multiple processor units already exist for a long time; however, they were rarely used in consumer systems. Today, all major CPU vendors are selling multicore processors. Nowadays, multicore processors are being used for everything from servers to consumer computers and even in smartphones. Because of this proliferation of multicore processors, writing multithreaded applications is becoming more and more important. A professional C++ programmer needs to know how to write correct

multithreaded code to take full advantage of all the available processor units. Writing multithreaded applications used to rely on platform- and operating system-specific APIs. This made it difficult to write platform-independent multithreaded code. C++11 solved this problem by including a standard threading library.

Multithreaded programming is a complicated subject. This chapter introduces you to multithreaded programming using the standard threading library, but it cannot go into all details due to space constraints. There are entire books written about developing multithreaded programs. If you are interested in more details, consult one of the references in the multithreading section in Appendix B.

If your compiler does not support the standard threading library, you might use other third-party libraries that try to make multithreaded programming more platform independent, such as the `pthreads` library and the `boost::thread` library. However, because they are not part of the C++ standard, they are not discussed in this book.

INTRODUCTION

Multithreaded programming allows you to perform multiple calculations in parallel. This way you can take advantage of the multiple processor units inside most systems these days. Years ago, the processor market was racing for the highest frequency, which is perfect for single-threaded applications. Around 2005, this race stopped due to a combination of power management and heat management problems. Today, the processor market is racing toward the most cores on a single processor chip. Dual- and quad-core processors are already common at the time of this writing, and announcements have already been made about 12-, 16-, 32-, and even 80-core processors.

Similarly, if you look at the processors on graphics cards, called GPUs, you'll see that they are massively parallel processors. Today, high-end graphics cards have more than 2,000 cores, a number that will increase rapidly. These graphics cards are used not only for gaming anymore, but also to perform computationally intense tasks. Examples are image and video manipulation, protein folding (useful for discovering new drugs), processing signals as part of the SETI project (Search for Extra-Terrestrial Intelligence), and so on.

C++98/03 did not have support for multithreaded programming, and you had to resort to third-party libraries or to the multithreading APIs of your target operating system. Since C++11 included a standard multithreading library, it became easier to write cross-platform multithreaded applications. The current C++ standard targets only CPUs and not GPUs. This might change in the future.

There are two reasons to start writing multithreaded code. First, if you have a computational problem and you manage to separate it into small pieces that can be run in parallel independently from each other, you can expect a huge performance boost running it on multiple processor units. Second, you can modularize computations along orthogonal axes; for example, doing long computations in a thread instead of blocking the GUI thread, so the user interface remains responsive while a long computation occurs in the background.

Figure 23-1 shows an example of a problem perfectly suited to run in parallel. An example could be the processing of pixels of an image by an algorithm that does not require information about neighboring pixels. The algorithm could split the image into four parts. On a single-core processor,

each part is processed sequentially; on a dual-core processor, two parts are processed in parallel; and on a quad-core processor, four parts are processed in parallel, resulting in an almost linear scaling of the performance with the number of cores.

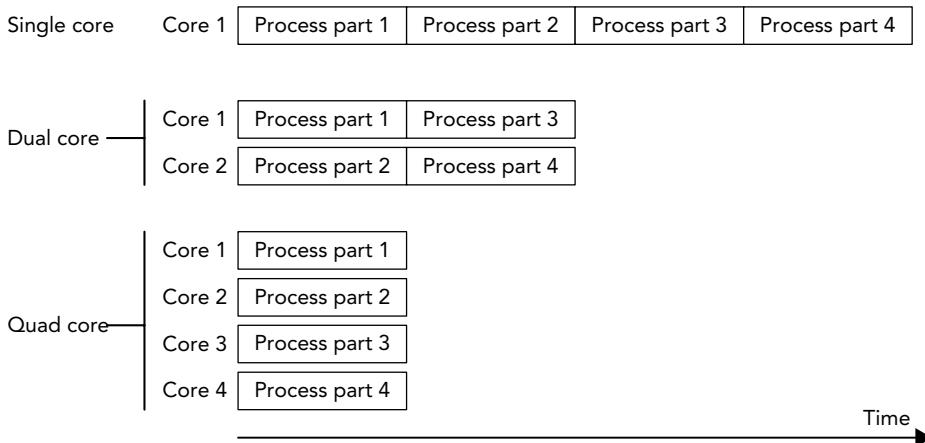


FIGURE 23-1

Of course, it's not always possible to split the problem into parts that can be executed independently of each other in parallel. But often it can be made parallel at least partially, resulting in a performance increase. A difficult part in multithreaded programming is to make your algorithm parallel, which is highly dependent on the type of your algorithm. Other difficulties are preventing race conditions, deadlocks, tearing, and keeping cache coherency in mind. These are discussed in the following sections. They can all be solved using atomics or explicit synchronization mechanisms, as discussed later in this chapter.

WARNING *To prevent these multithreading problems, try to design your programs so that multiple threads need not read and write to shared memory. Or, use a synchronization method as described in the Mutual Exclusion section, or atomic operations described in the Atomic Operations Library section.*

Race Conditions

Race conditions can occur when multiple threads want to read/write to a shared memory location. For example, suppose you have a shared variable and one thread increments this value while another thread decrements it. Incrementing and decrementing the value means that the current value needs to be retrieved from memory, incremented or decremented, and stored back in memory. On older architectures, such as PDP-11 and VAX, this used to be implemented with an `INC` processor instruction, which was atomic. On modern x86 processors, the `INC` instruction is not atomic anymore, meaning that other instructions could be executed in the middle of this operation, which might cause the code to retrieve a wrong value.

The following table shows the result when the increment is finished before the decrement starts, and assumes that the initial value is 1:

THREAD 1 (INCREMENT)	THREAD 2 (DECREMENT)
load value (value = 1)	
increment value (value = 2)	
store value (value = 2)	
	load value (value = 2)
	decrement value (value = 1)
	store value (value = 1)

The final value stored in memory is 1. When the decrement thread is finished before the increment thread starts, the final value is also 1, as seen in the following table:

THREAD 1 (INCREMENT)	THREAD 2 (DECREMENT)
	load value (value = 1)
	decrement value (value = 0)
	store value (value = 0)
load value (value = 0)	
increment value (value = 1)	
store value (value = 1)	

However, when the instructions get interleaved, the result is different:

THREAD 1 (INCREMENT)	THREAD 2 (DECREMENT)
load value (value = 1)	
increment value (value = 2)	
	load value (value = 1)
	decrement value (value = 0)
store value (value = 2)	
	store value (value = 0)

The final result in this case is 0. In other words, the effect of the increment operation is lost. This is a race condition.

Deadlocks

If you opt to solve a race condition by using a synchronization method, such as mutual exclusion, you might run into another common problem with multithreaded programming: *deadlocks*. Deadlocks are threads blocking indefinitely because they are waiting to acquire access to resources currently locked by other blocked threads. For example, suppose you have two threads and two resources, A and B. Both threads require a lock on both resources, but they acquire the locks in different order. The following table shows this situation in pseudo code:

THREAD 1	THREAD 2
Lock A	Lock B
Lock B	Lock A
// ... compute	// ... compute
Release B	Release A
Release A	Release B

Now, imagine that the code in the two threads is executed in the following order:

- Thread 1: Lock A
- Thread 2: Lock B
- Thread 1: Lock B (waits, because lock held by Thread 2)
- Thread 2: Lock A (waits, because lock held by Thread 1)

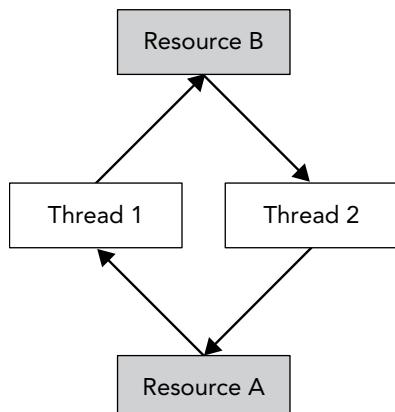


FIGURE 23-2

Both threads are now waiting indefinitely in a deadlock situation. Figure 23-2 shows a graphical representation of this deadlock situation. Thread 1 is holding a lock on resource A and is waiting to get a lock on resource B. Thread 2 is holding a lock on resource B and is waiting to get a lock on resource A. In this graphical representation, you see a cycle that depicts the deadlock situation. Both threads will wait indefinitely.

It's best to always acquire locks in the same order to avoid these kinds of deadlocks. You can also include mechanisms in your program to break these kinds of deadlocks. One possible solution is to try for a certain time to acquire a lock on a resource. If the lock could not be obtained within a certain time interval, the thread stops waiting and possibly releases other locks it is currently holding. The thread might then sleep for a little bit and try again later to acquire all the resources it needs. This method might give other threads the opportunity to acquire necessary locks and continue their execution. Whether this method works or not depends heavily on your specific deadlock case.

Instead of using a workaround as described in the previous paragraph, you should try to avoid any possible deadlock situation altogether. If you need to acquire multiple locks, the recommended way is to use the standard `std::lock()` or `std::try_lock()` functions described later in the section on mutual exclusion. These functions obtain or try to obtain a lock on several resources, doing their best to prevent deadlocks.

Tearing

Tearing means that part of your data has been written to memory, while part hasn't been written yet. If another thread reads that data at that exact moment it sees inconsistent data.

Cache Coherency

Cache coherency is important to keep in mind. If one thread writes a piece of data, that thread immediately sees this new data, but this does not mean that all threads see this new data immediately! CPUs have caches and the cache structure on multicore processors is complicated. If one core modifies your data, it is changed immediately in its cache; but, this change is not immediately visible to cores using a different cache. So, even simple data types, such as Booleans, need to be synchronized when reading and writing to them from multiple threads.

THREADS

The C++ threading library, defined in the `<thread>` header file, makes it very easy to launch new threads. Specifying what needs to be executed in the new thread can be done in several ways. You can let the new thread execute a global function, the `operator()` of a function object, a lambda expression, or even a member function of an instance of some class. The following sections give small examples of all these methods.

Thread with Function Pointer

Functions such as `CreateThread()`, `_beginthread()`, and so on, on Windows, and `pthread_create()` with the `pthread` library, require that the thread function has only one parameter. On the other

hand, a function that you want to use with the standard C++ `std::thread` class can have as many parameters as you want.

Suppose you have a `counter()` function accepting two integers: the first representing an ID and the second representing the number of iterations that the function should loop. The body of the function is a single loop that loops the given number of iterations. On each iteration, a message is printed to standard output:

```
void counter(int id, int numIterations)
{
    for (int i = 0; i < numIterations; ++i) {
        cout << "Counter " << id << " has value ";
        cout << i << endl;
    }
}
```

You can launch multiple threads executing this function using `std::thread`. You can create a thread `t1`, executing `counter()` with arguments 1 and 6 as follows:

```
thread t1(counter, 1, 6);
```

The constructor of the `thread` class is a variadic template, which means that it accepts any number of arguments. Variadic templates are discussed in detail in Chapter 21. The first argument is the name of the function to execute in the new thread. The subsequent variable number of arguments are passed to this function when execution of the thread starts.

The following code launches two threads executing the `counter()` function. After launching the threads, `main()` calls `join()` on both threads. This is to make sure that the main thread keeps running until both threads are finished. A call to `t1.join()` blocks until the thread `t1` is finished. Without these two `join()` calls, the `main()` function would finish immediately after launching the two threads. This will trigger the application to shut down; causing all other threads spawned by the application to be terminated as well, whether these threads are finished or not.

WARNING *These `join()` calls are necessary in these small examples. In real-world applications, you should avoid using `join()`, because it causes the thread calling `join()` to block. Often there are better ways. For example, in a GUI application, a thread that finishes can post a message to the UI thread. The UI thread itself has a message loop processing messages like mouse moves, button clicks, and so on. This message loop can also receive messages from threads, and you can react to them however you want, all without blocking the UI thread with a `join()` call.*

```
#include <iostream>
#include <thread>
using namespace std;
int main()
{
    thread t1(counter, 1, 6);
    thread t2(counter, 2, 4);
```

```

    t1.join();
    t2.join();
}

```

A possible output of this example looks as follows:

```

Counter 2 has value 0
Counter 1 has value 0
Counter 1 has value 1
Counter 1 has value 2
Counter 1 has value 3
Counter 1 has value 4
Counter 1 has value 5
Counter 2 has value 1
Counter 2 has value 2
Counter 2 has value 3

```

The output on your system will be different and it will most likely be different every time you run it. This is because two threads are executing the `counter()` function at the same time, so the output depends on the number of processing cores in your system and on the thread scheduling of the operating system.

By default, accessing `cout` from different threads is thread-safe and without any data races, unless you have called `cout.sync_with_stdio(false)` before the first output or input operation. However, even though there are no data races, output from different threads can still be interleaved! This means that the output of the previous example can be mixed together as in the following:

```

Counter Counter 2 has value 0
1 has value 0
Counter 1 has value 1
Counter 1 has value 2

```

Instead of:

```

Counter 1 has value 0
Counter 2 has value 0
Counter 1 has value 1
Counter 1 has value 2

```

This can be fixed using synchronization methods, which are discussed later in this chapter.

NOTE *Thread function arguments are always copied into some internal storage for the thread. Use `std::ref()` from the `<functional>` header to pass them by reference.*

Thread with Function Object

The previous section demonstrated how to create a thread and tell it to run a specific function in the new thread by passing a pointer to the function to execute. You can also use a function object,

as shown in the following example. With the function pointer technique, the only way to pass information to the thread is by passing arguments to the function. With function objects, you can add member variables to your function object class, which you can initialize and use however you want. The example first defines a class called `Counter`, which has two member variables: an ID and the number of iterations for the loop. Both variables are initialized with the constructor. To make the `Counter` class a function object, you need to implement `operator()`, as discussed in Chapter 17. The implementation of `operator()` is the same as the `counter()` function in the previous section:

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : mId(id), mNumIterations(numIterations)
    {
    }
    void operator()() const
    {
        for (int i = 0; i < mNumIterations; ++i) {
            cout << "Counter " << mId << " has value ";
            cout << i << endl;
        }
    }
private:
    int mId;
    int mNumIterations;
};
```

Three methods for initializing threads with a function object are demonstrated in the following `main()`. The first uses the uniform initialization syntax. You create an instance of `Counter` with its constructor arguments and give it to the thread constructor between curly braces.

The second defines a named instance of `Counter` and gives this named instance to the constructor of the thread class.

The third looks similar to the first; it creates an instance of `Counter` and gives it to the constructor of the thread class, but uses parentheses instead of curly braces. The ramifications of this are discussed after the code.

```
int main()
{
    // Using uniform initialization syntax
    thread t1{ Counter{ 1, 20 } };
    // Using named variable
    Counter c(2, 12);
    thread t2(c);
    // Using temporary
    thread t3(Counter(3, 10));
    // Wait for threads to finish
    t1.join();
    t2.join();
    t3.join();
}
```

If you compare the creation of `t1` with the creation of `t3`, it looks like the only difference seems to be that the first method uses curly braces while the third method uses parentheses. However, when your function object constructor doesn't require any parameters, the third method as written above will not work. For example:

```
class Counter
{
public:
    Counter() {}
    void operator()() const { /* Omitted for brevity */ }
};

int main()
{
    thread t1(Counter());      // Error!
    t1.join();
}
```

This results in a compilation error because C++ interprets the first line in `main()` as a declaration of a function called `t1`, which returns a `thread` object and accepts a pointer to a function without parameters returning a `Counter` object. For this reason, it's recommended to use the uniform initialization syntax:

```
thread t1{ Counter{} }; // OK
```

If your compiler does not support uniform initialization, you have to add an extra set of parentheses to prevent the compiler from interpreting the line as a function declaration:

```
thread t1((Counter())); // OK
```

NOTE *Function objects are always copied into some internal storage for the thread. If you want to execute `operator()` on a specific instance of your function object instead of copying it, you should use `std::ref()` from the `<functional>` header to pass your instance by reference.*

Thread with Lambda

Lambda expressions fit nicely with the standard C++ threading library, as demonstrated in the following example:

```
int main()
{
    int id = 1;
    int numIterations = 5;
    thread t1([id, numIterations] {
        for (int i = 0; i < numIterations; ++i) {
            cout << "Counter " << id << " has value ";
            cout << i << endl;
        }
    });
    t1.join();
}
```

Thread with Member Function

You can also specify a member function of a class to be executed in a thread. The following example defines a basic Request class with a `process()` method. The `main()` function creates an instance of the Request class and launches a new thread, which executes the `process()` member function of the Request instance, `req`:

```
class Request
{
public:
    Request(int id) : mId(id) { }
    void process()
    {
        cout << "Processing request " << mId << endl;
    }
private:
    int mId;
};

int main()
{
    Request req(100);
    thread t{ &Request::process, &req };
    t.join();
}
```

With this technique you are executing a method on a specific object in a separate thread. If other threads are accessing the same object, you need to make sure this happens in a thread-safe way to avoid race conditions. Mutual exclusion, discussed later in this chapter, can be used as synchronization mechanism to make it thread-safe.

Thread Local Storage

The standard supports the concept of *thread local storage*. With a keyword called `thread_local`, you can mark any variable as thread local, which means that each thread will have its own unique copy of the variable and it will last for the entire duration of the thread. For each thread, the variable is initialized exactly once. For example, in the following code, every thread shares one-and-only-one copy of `k`, while each thread has its own unique copy of `n`:

```
thread_local int n;
int k;
void doWork()
{
    // perform some computation
}
```

Note that if the `thread_local` variable is declared in the scope of a function, its behavior is as if it were declared `static`, except that every thread has its own unique copy and is initialized exactly once per thread, no matter how many times that function is called in that thread.

Cancelling Threads

The standard does not include any mechanism for cancelling a running thread from inside another thread. The best way to achieve this is to provide some communication mechanism that the two threads agree upon. The simplest mechanism is to have a shared variable, which the target thread checks periodically to determine if it should terminate. Other threads can set this shared variable to indirectly instruct the thread to shut down. Care has to be taken to avoid race conditions and cache coherency problems with reading and writing to this shared variable. Atomic variables or condition variables, both discussed later in this chapter, can help avoid these problems.

Retrieving Results from Threads

As you saw in the previous examples, launching a new thread is pretty easy. However, in most cases you are probably interested in results produced by the thread. For example, if your thread performs some mathematical calculations, you really would like to get the results out of the thread once the thread is finished. One way is to pass a pointer or reference to a result variable to the thread in which the thread stores the results. Another method is to store the results inside a class member variable of a function object, which you can retrieve later once the thread has finished executing.

However, there is another and easier method to obtain a result from threads: *futures*. They also make it easier to handle errors that occur inside your threads. Futures are discussed later in this chapter.

Copying and Rethrowing Exceptions

The whole exception mechanism in C++ works perfectly, as long as it stays within one single thread. Every thread can throw its own exceptions, but they need to be caught within their own thread. Exceptions thrown in one thread cannot be caught in another thread. This introduces quite a few problems when you would like to use exception handling in combination with multithreaded programming.

Without the standard threading library it's very difficult if not impossible to gracefully handle exceptions across threads. The standard threading library solves this issue with the following exception-related functions. These functions not only work with `std::exception`, but with all kinds of exceptions, ints, strings, custom exceptions, and so on:

```
exception_ptr current_exception() noexcept;
```

This function is intended to be called from inside a catch block, and returns an `exception_ptr` object that refers to the exception currently being handled, or a copy of the currently handled exception, or a null `exception_ptr` object if no exception is being handled. This referenced exception object remains valid for as long as there is an object of type `exception_ptr` that is referencing it. `exception_ptr` is of type `NullablePointer`, which means it can easily be tested with a simple `if` statement, as the example later in this section demonstrates:

```
[[noreturn]] void rethrow_exception(exception_ptr p);
```

This function rethrows the exception referenced by the `exception_ptr` parameter. Rethrowing the referenced exception does not have to be done in the same thread that generated the referenced exception in the first place, which makes this feature perfectly suited for handling exceptions across different threads. The `[[noreturn]]` attribute makes it clear that this function never returns normally. Attributes are introduced in Chapter 10.

```
template<class E> exception_ptr make_exception_ptr(E e) noexcept;
```

This function creates an `exception_ptr` object that refers to a copy of the given exception object. This is basically a shorthand notation for the following code:

```
try {
    throw e;
} catch(...) {
    return current_exception();
}
```

Let's see how handling exceptions across different threads can be implemented using these features. The following code defines a function that does some work and throws an exception. This function will ultimately be running in a separate background thread:

```
void doSomeWork()
{
    for (int i = 0; i < 5; ++i) {
        cout << i << endl;
    }
    cout << "Thread throwing a runtime_error exception..." << endl;
    throw runtime_error("Exception from thread");
}
```

The following `threadFunc()` function wraps the call to the preceding function in a `try/catch` block, catching all exceptions that `doSomeWork()` might throw. A single argument is supplied to `threadFunc()`, which is of type `exception_ptr&`. Once an exception is caught, the function `current_exception()` is used to get a reference to the exception being handled, which is then assigned to the `exception_ptr` parameter. After that, the thread exits normally:

```
void threadFunc(exception_ptr& err)
{
    try {
        doSomeWork();
    } catch (...) {
        cout << "Thread caught exception, returning exception..." << endl;
        err = current_exception();
    }
}
```

The following `doWorkInThread()` function is called from within the main thread. Its responsibility is to create a new thread and start executing `threadFunc()` in it. A reference to an object of type `exception_ptr` is given as argument to `threadFunc()`. Once the thread is created, the `doWorkInThread()` function waits for the thread to finish by using the `join()` method, after which

the error object is examined. Since `exception_ptr` is of type `NullablePointer`, you can easily check it using an `if` statement. If it's a non-null value, the exception is rethrown in the current thread, which is the main thread in this example. By rethrowing the exception in the main thread, the exception has been transferred from one thread to another thread.

```
void doWorkInThread()
{
    exception_ptr error;
    // Launch background thread
    thread t{ threadFunc, ref(error) };
    // Wait for thread to finish
    t.join();
    // See if thread has thrown any exception
    if (error)
    {
        cout << "Main thread received exception, rethrowing it..." << endl;
        rethrow_exception(error);
    }
    else
        cout << "Main thread did not receive any exception." << endl;
}
```

The `main()` function is pretty straightforward. It calls `doWorkInThread()` and wraps the call in a `try/catch` block to catch exceptions thrown by any thread spawned by `doWorkInThread()`:

```
int main()
{
    try {
        doWorkInThread();
    } catch (const exception& e) {
        cout << "Main function caught: '" << e.what() << "'" << endl;
    }
}
```

The output is as follows:

```
0
1
2
3
4
Thread throwing a runtime_error exception...
Thread caught exception, returning exception...
Main thread received exception, rethrowing it...
Main function caught: 'Exception from thread'
```

To keep this example compact and easier to understand, the `doWorkInThread()` function is using `join()` to block and wait until the thread is finished. Of course, in real-world applications you do not want to block your main thread. For example, in a GUI application, you might let `threadFunc()` send a message to the UI thread with, as argument, a copy of the result of `current_exception()`.

ATOMIC OPERATIONS LIBRARY

Atomic types allow atomic access, which means that concurrent reading and writing without additional synchronization is allowed. Without atomic operations, incrementing a variable is not thread-safe because the compiler first loads the value from memory into a register, increments it, and then stores the result back in memory. Another thread might touch the same memory during this increment operation, which is a race condition. For example, the following code is not thread-safe and contains a race condition. This type of race condition is discussed in the beginning of this chapter:

```
int counter = 0;      // Global variable
++counter;           // Executed in multiple threads
```

To make this thread-safe without explicitly using any locks, use an atomic type:

```
atomic<int> counter(0); // Global variable
++counter;              // Executed in multiple threads
```

You need to include the `<atomic>` header to use these atomic types. The standard defines named integral atomic types for all primitive types. The following table lists a few:

NAMED ATOMIC TYPE	EQUIVALENT ATOMIC TYPE
<code>atomic_bool</code>	<code>atomic<bool></code>
<code>atomic_char</code>	<code>atomic<char></code>
<code>atomic_uchar</code>	<code>atomic<unsigned char></code>
<code>atomic_int</code>	<code>atomic<int></code>
<code>atomic_uint</code>	<code>atomic<unsigned int></code>
<code>atomic_long</code>	<code>atomic<long></code>
<code>atomic_ulong</code>	<code>atomic<unsigned long></code>
<code>atomic_llong</code>	<code>atomic<long long></code>
<code>atomic_ullong</code>	<code>atomic<unsigned long long></code>
<code>atomic_wchar_t</code>	<code>atomic<wchar_t></code>

When accessing a piece of data from multiple threads, atomics also solve other problems such as cache coherence, memory ordering, compiler optimizations, and so on. Basically, it's virtually never safe to read and write to the same piece of data from multiple threads without using atomics or explicit synchronization mechanisms.

Atomic Type Example

This section explains in more detail why you should use atomic types. Suppose you have a function called `func()` that increments an integer given as a reference parameter in a loop. This code uses

`std::this_thread::sleep_for()` to introduce a small delay in each loop. The argument to `sleep_for()` is a `std::chrono::duration`, explained in Chapter 19.

```
void func(int& counter)
{
    for (int i = 0; i < 100; ++i) {
        ++counter;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

Now, you would like to run several threads in parallel, all executing this `func()` function. By implementing this naively without atomic types or without any kind of thread synchronization, you introduce a race condition. The following `main()` function launches 10 threads after which it waits for all threads to finish by calling `join()` on each thread.

```
int main()
{
    int counter = 0;
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.push_back(std::thread{ func, std::ref(counter) });
    }
    for (auto& t : threads) {
        t.join();
    }
    std::cout << "Result = " << counter << std::endl;
}
```

Because `func()` increments the integer 100 times, and `main()` launches 10 background threads, each of which executes `func()`, the expected result is 1,000. If you execute this program several times, you might get the following output but with different values:

```
Result = 982
Result = 977
Result = 984
```

This code is clearly showing race condition behavior. In this example you can use an atomic type to fix this. The following code highlights the required changes:

```
#include <atomic>
void func(std::atomic<int>& counter)
{
    for (int i = 0; i < 100; ++i) {
        ++counter;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
int main()
{
    std::atomic<int> counter(0);
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.push_back(std::thread{ func, std::ref(counter) });
    }
}
```

```

        for (auto& t : threads) {
            t.join();
        }
        std::cout << "Result = " << counter << std::endl;
    }
}

```

The changes add the `<atomic>` header file, and change the type of the shared counter to `std::atomic<int>` instead of `int`. When you run this modified version, you always get 1,000 as the result:

```

Result = 1000
Result = 1000
Result = 1000

```

Without explicitly adding any locks to the code, it is now thread-safe and race-condition free because the `++counter` operation on an atomic type loads the value, increments the value, and stores the value in one atomic transaction, which cannot be interrupted.

However, there is a new problem with this new code; a performance problem. You should try to minimize the amount of synchronization, either atomics or explicit synchronization, because it lowers performance. For this simple example, the best and recommended solution is to let `func()` calculate its result in a local variable, and only after the loop add it to the `counter` reference. Note that it is still required to use an atomic, because you are still writing to `counter` from multiple threads.

```

#include <atomic>
void func(std::atomic<int>& counter)
{
    int result = 0;
    for (int i = 0; i < 100; ++i) {
        ++result;
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
    counter += result;
}

```

Atomic Operations

The standard defines a number of atomic operations. This section describes a few of those operations. For a full list, consult a Standard Library Reference; for example, <http://www.cppreference.com/> or <http://www.cplusplus.com/reference/>.

A first example of an atomic operation is the following:

```
bool atomic_compare_exchange_strong(atomic<C>* object, C* expected, C desired);
```

It can also be called as a member of `atomic<C>`:

```
bool atomic<C>::compare_exchange_strong(C* expected, C desired);
```

The logic implemented atomically by this operation is as follows in pseudo-code:

```

if (*object == *expected) {
    *object = desired;
}

```

```

        return true;
    } else {
        *expected = *object;
        return false;
    }
}

```

A second example is `atomic<T>::fetch_add()`, which works for integral atomic types and fetches the current value of the atomic type, adds the given increment to the atomic value, and returns the original non-incremented value. For example:

```

atomic<int> value(10);
cout << "Value = " << value << endl;
int fetched = value.fetch_add(4);
cout << "Fetched = " << fetched << endl;
cout << "Value = " << value << endl;

```

If no other threads are touching the contents of the `fetched` and `value` variables, the output is as follows:

```

Value = 10
Fetched = 10
Value = 14

```

Atomic integral types support the following atomic operations: `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, `fetch_xor()`, `++, --, +=, -=, &=, ^=, and |=`. Atomic pointer types support `fetch_add()`, `fetch_sub()`, `++, --, +=, and -=`.

Most of the atomic operations can accept an extra parameter specifying the memory ordering that you would like. For example:

```
T atomic<T>::fetch_add(T value, memory_order = memory_order_seq_cst);
```

You may change the default `memory_order`. The standard provides: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`, all defined in the `std` namespace. However, it is rare that you will want to use them instead of the default. While another memory order may perform better than the default, according to some metric, if you use them slightly wrong you will again introduce race conditions or other difficult-to-track threading-related problems. If you do want to know more about memory ordering, consult one of the multithreading references in Appendix B.

MUTUAL EXCLUSION

If you are writing multithreaded applications, you have to be sensitive to sequencing of operations. If your threads read and write shared data, this can be a problem. There are many ways to avoid this problem, such as never actually sharing data between threads. However, if you can't avoid sharing data, you must provide for synchronization so that only one thread at a time can change the data.

Scalars such as Booleans and integers can often be synchronized properly with atomic operations described earlier; but, when your data is more complex, and you need to use that data from multiple threads, you must provide explicit synchronization.

The standard library has support for mutual exclusion in the form of *mutex* and *lock* classes. These can be used to implement synchronization between threads and are discussed in the next sections.

Mutex Classes

Mutex stands for mutual exclusion. The mutual exclusion classes are all defined in the `<mutex>` header file and are in the `std` namespace. The basic mechanism of using a mutex is as follows:

- A thread that wants to use (read/write) memory shared with other threads tries to lock a mutex object. If another thread is currently holding this lock, the new thread that wants to gain access blocks until the lock is released, or until a timeout interval expires.
- Once the thread has obtained the lock, it is free to use the shared memory. Of course, this assumes that all threads that want to use the shared data all correctly acquire a lock on the mutex.
- After the thread is finished with reading/writing to the shared memory, it releases the lock to give some other thread an opportunity to obtain the lock to the shared memory. If two or more threads are waiting on the lock, there are no guarantees as to which thread is granted the lock and thus allowed to proceed.

The standard provides *non-timed mutex* and *timed mutex* classes.

Non-Timed Mutex Classes

The library has two non-timed mutex classes: `std::mutex` and `std::recursive_mutex`. Each supports the following methods:

- `lock()`: The calling thread tries to obtain the lock and blocks until the lock has been acquired. It blocks indefinitely. If there is a desire to limit the amount of time the thread blocks, you should use a timed mutex, discussed in the next section.
- `try_lock()`: The calling thread tries to obtain the lock. If the lock is currently held by another thread, the call returns immediately. If the lock has been obtained, `try_lock()` returns `true`, otherwise it returns `false`.
- `unlock()`: Releases the lock held by the calling thread, making it available for another thread.

`std::mutex` is a standard mutual exclusion class with exclusive ownership semantics. There can be only one thread owning the mutex. If another thread wants to obtain ownership of this mutex, it either blocks when using `lock()`, or fails when using `try_lock()`. A thread already having ownership of a `std::mutex` is not allowed to call `lock()` or `try_lock()` again on that mutex. This might lead to a deadlock!

`std::recursive_mutex` behaves almost identically to `std::mutex`, except that a thread already having ownership of a recursive mutex is allowed to call `lock()` or `try_lock()` again on the same recursive mutex. The calling thread should call the `unlock()` method as many times as it obtained a lock on the recursive mutex.

Timed Mutex Classes

The library provides three timed mutex classes: `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_timed_mutex`; all support the normal `lock()`, `try_lock()`, and `unlock()` methods. Additionally, they support the following:

- `try_lock_for(rel_time)`: The calling thread tries to obtain the lock for a certain relative time. If the lock could not be obtained after the given timeout, the call fails and returns `false`. If the lock could be obtained within the timeout, the call succeeds and returns `true`.
- `try_lock_until(abs_time)`: The calling thread tries to obtain the lock until the system time equals or exceeds the specified absolute time. If the lock could be obtained before this time, the call returns `true`. If the system time passes the given absolute time, the function stops trying to obtain the lock and returns `false`.

A thread already having ownership of a `timed_mutex` is not allowed to call one of the previous lock calls again on that mutex. This might lead to a deadlock!

`recursive_timed_mutex` behaves almost identically to `timed_mutex`, except that a thread already having ownership of a recursive mutex is allowed to call one of the previous lock calls again on the same mutex. The calling thread should call the `unlock()` method as many times as it obtained a lock on the recursive mutex.

The `shared_timed_mutex` class supports the concept of *shared lock ownership*, also known as *readers-writers lock*. A thread can either get *exclusive ownership* or *shared ownership* of the lock. Exclusive ownership, also known as a *write lock*, can be acquired only when there are no other threads having exclusive or shared ownership. Shared ownership, also known as a *read lock*, can be acquired if there is no other thread having exclusive ownership, but other threads are allowed to have acquired shared ownership. The `shared_timed_mutex` class supports `lock()`, `try_lock()`, `try_lock_for()`, `try_lock_until()`, and `unlock()`, all discussed earlier. These methods acquire and release exclusive locks. Additionally they have the following shared ownership-related methods:

- `lock_shared()`: The calling thread tries to obtain the shared ownership lock and blocks until the lock has been acquired.
- `try_lock_shared()`: The calling thread tries to obtain the shared ownership lock. If an exclusive lock is currently held by another thread, the call returns immediately. If the lock has been obtained, `try_lock()` returns `true`, otherwise it returns `false`.
- `try_lock_shared_for(rel_time)`: The calling thread tries to obtain the shared ownership lock for a certain relative time. If the lock could not be obtained after the given timeout, the call fails and returns `false`. If the lock could be obtained within the timeout, the call succeeds and returns `true`.
- `try_lock_shared_until(abs_time)`: The calling thread tries to obtain the shared ownership lock until the system time equals or exceeds the specified absolute time. If the lock could be obtained before this time, the call returns `true`. If the system time passes the given absolute time, the function stops trying to obtain the lock and returns `false`.
- `unlock_shared()`: Releases shared ownership.

A thread already having a lock on a `shared_timed_mutex` is not allowed to try to acquire a second lock on that mutex. This might lead to a deadlock!

WARNING *Do not manually call one of the previously discussed lock and unlock methods on any of the mutex classes. Mutex locks are resources, and, as all resources, they almost exclusively should be acquired using the RAII (Resource Acquisition Is Initialization) paradigm. The standard defines a number of RAII lock classes, discussed in the next section. Using them is critical to avoid deadlocks. They automatically unlock a mutex when a lock object goes out of scope, so you don't need to manually call unlock() at the right time.*

Locks

A `lock` class is a RAII class that makes it easier to correctly obtain and release a lock on a mutex; the destructor of the lock class automatically releases the associated mutex. The standard defines three types of locks: `std::lock_guard`, `std::unique_lock`, and `std::shared_lock`.

lock_guard

`lock_guard` is a simple lock with two constructors.

- `explicit lock_guard(mutex_type& m);`

A constructor accepting a reference to a mutex. This one tries to obtain a lock on the mutex and blocks until the lock is obtained. The keyword `explicit` for constructors is discussed in Chapter 8.

- `lock_guard(mutex_type& m, adopt_lock_t);`

A constructor accepting a reference to a mutex and an instance of the `std::adopt_lock_t` struct. The lock assumes that the calling thread already has obtained a lock on the referenced mutex and will manage this lock.

unique_lock

`std::unique_lock` is a more sophisticated lock that allows you to defer lock acquisition until later in the execution, long after the declaration. You can use the `owns_lock()` method to see if the lock has been acquired. A `unique_lock` also has a `bool` conversion operator, which can be used to check if the lock has been acquired. An example of using this conversion operator is given later in this chapter in the section “Using Timed Locks.” `unique_lock` has several constructors:

- `explicit unique_lock(mutex_type& m);`

A constructor accepting a reference to a mutex. This one tries to obtain a lock on the mutex and blocks until the lock is obtained.

- `unique_lock(mutex_type& m, defer_lock_t) noexcept;`

A constructor accepting a reference to a mutex and an instance of the `std::defer_lock_t` struct. The `unique_lock` stores the reference to the mutex, but does not immediately try to obtain a lock. A lock can be obtained later.

- `unique_lock(mutex_type& m, try_to_lock_t);`

A constructor accepting a reference to a mutex and an instance of the `std::try_to_lock_t` struct. The lock tries to obtain a lock to the referenced mutex, but if it fails it does not block.

- `unique_lock(mutex_type& m, adopt_lock_t);`

A constructor accepting a reference to a mutex and an instance of the `std::adopt_lock_t` struct. The lock assumes that the calling thread already has obtained a lock on the referenced mutex and will manage this lock.

- `template <class Clock, class Duration>`

- `unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);`

A constructor accepting a reference to a mutex and an absolute time. The constructor tries to obtain a lock until the system time passes the given absolute time. The Chrono library is discussed in Chapter 19.

- `template <class Rep, class Period>`

- `unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);`

A constructor accepting a reference to a mutex and a relative time. The constructor tries to get a lock on the mutex with the given relative timeout.

The `unique_lock` class also has the following methods: `lock()`, `try_lock()`, `try_lock_for()`, `try_lock_until()`, and `unlock()`, which behave as explained in the section on timed mutex classes earlier in this chapter.

shared_lock

The `shared_lock` class has the same type of constructors and the same methods as `unique_lock`. The difference is that the `shared_lock` class calls the shared ownership related methods on the underlying shared mutex. Thus, the methods of `shared_lock` are called `lock()`, `try_lock()`, and so on, but on the underlying shared mutex they call `lock_shared()`, `try_lock_shared()`, and so on. This is done so that `shared_lock` has the same interface as `unique_lock`, and can be used as a stand-in replacement for `unique_lock` but acquires a shared lock instead of an exclusive lock.

Acquiring Multiple Locks at Once

C++ has two generic lock functions that you can use to obtain locks on multiple mutex objects at once without the risk of creating deadlocks. Both are defined in the `std` namespace, and both are variadic template functions, discussed in Chapter 21.

```
template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

This generic function locks all the given mutex objects in an unspecified order without the risk of deadlocks. If one of the mutex lock calls throws an exception, `unlock()` is called on all locks that have already been obtained.

```
template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

`try_lock()` tries to obtain a lock on all the given mutex objects by calling `try_lock()` on each of them in sequence. It returns -1 if all calls to `try_lock()` succeed. If any `try_lock()` fails, `unlock()` is called on all locks that have already been obtained, and the return value is the zero-based index of the parameter position of the mutex on which `try_lock()` failed.

The following example demonstrates how to use the generic `lock()` function. The `process()` function first creates two locks, one for each mutex, and gives an instance of `std::defer_lock_t` as a second argument to tell `unique_lock` not to acquire the lock during construction. The call to `lock()` then acquires both locks without the risk of deadlocks:

```
mutex mut1;
mutex mut2;
void process()
{
    unique_lock<mutex> lock1(mut1, defer_lock_t());
    unique_lock<mutex> lock2(mut2, defer_lock_t());
    lock(lock1, lock2);
    // Locks acquired
}
int main()
{
    process();
}
```

std::call_once

You can use `std::call_once()` in combination with `std::once_flag` to make sure a certain function or method is called exactly one time no matter how many threads try to call `call_once()`. Only one `call_once()` invocation actually calls the given function or method; this invocation is called the *effective call_once()*. This effective invocation on a specific `once_flag` instance finishes before all other `call_once()` invocations on the same `once_flag` instance. Other threads calling `call_once()` on the same `once_flag` instance block until the effective call is finished. Figure 23-3 illustrates this with three threads. Thread 1 performs the effective `call_once()` invocation, Thread 2 blocks until the effective invocation is finished, and Thread 3 doesn't block because the effective invocation from Thread 1 has already finished.

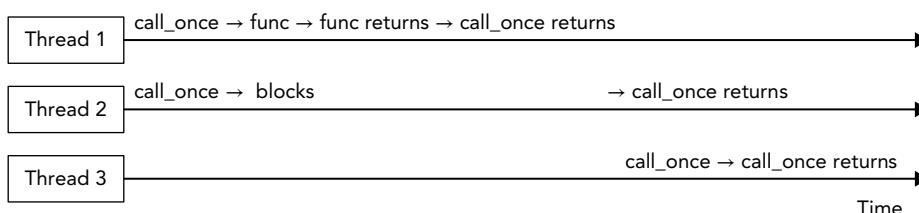


FIGURE 23-3

The following example demonstrates the use of `call_once()`. The example launches three threads running `processingFunction()` that uses some shared resources. These shared resources should be initialized only once by calling `initializeSharedResources()` once. To accomplish this, each thread calls `call_once()` with a global `once_flag`. The result is that only one thread executes `initializeSharedResources()`, and exactly one time. While this `call_once()` call is in progress, other threads block until `initializeSharedResources()` returns:

```
once_flag gOnceFlag;
void initializeSharedResources()
{
    // ... Initialize shared resources that will be used by multiple threads.
    cout << "Shared resources initialized." << endl;
}
void processingFunction()
{
    // Make sure the shared resources are initialized.
    call_once(gOnceFlag, initializeSharedResources);
    // ... Do some work, including using the shared resources
    cout << "Processing" << endl;
}
int main()
{
    // Launch 3 threads.
    vector<thread> threads(3);
    for (auto& t : threads) {
        t = thread{ processingFunction };
    }
    // Join on all threads
    for (auto& t : threads) {
        t.join();
    }
}
```

The output of this code is as follows:

```
Shared resources initialized.
Processing
Processing
Processing
```

Of course, in this example, you could call `initializeSharedResources()` once in the beginning of the `main()` function before the threads are launched; however, that wouldn't demonstrate the use of `call_once()`.

Examples Using Mutual Exclusion Objects

Thread-Safe Writing to Streams

Earlier in this chapter, in the section about threads, there is an example with a class called `Counter`. That example mentions that C++ streams are race-condition free by default, but that the output from multiple threads can be interleaved. To solve this interleaving issue, you can use a mutual exclusion object to make sure that only one thread at a time is reading/writing to the stream object.

The following example synchronizes all accesses to `cout` in the `Counter` class. For this, a static `mutex` object is added to the class. It should be static, because all instances of the class should use the same `mutex` instance. `lock_guard` is used to obtain a lock on the `mutex` before writing to `cout`. Changes compared to the earlier version are highlighted:

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : mId(id), mNumIterations(numIterations)
    {
    }
    void operator()() const
    {
        for (int i = 0; i < mNumIterations; ++i) {
            lock_guard<mutex> lock(mMutex);
            cout << "Counter " << mId << " has value ";
            cout << i << endl;
        }
    }
private:
    int mId;
    int mNumIterations;
    static mutex mMutex;
};
mutex Counter::mMutex;
```

This code creates a `lock_guard` instance on each iteration of the `for` loop. It is recommended to limit the time a lock is held as much as possible, otherwise you are blocking other threads for too long. For example, if the `lock_guard` instance would be created once right before the `for` loop, then you basically lose all multithreading in this code because one thread holds a lock for the entire duration of its `for` loop, and all other threads wait for this lock to be released.

Using Timed Locks

The following example demonstrates how to use a timed mutex. It is the same `Counter` class as before, but this time it uses a `timed_mutex` in combination with a `unique_lock`. A relative time of 200 milliseconds is given to the `unique_lock` constructor, causing it to try to obtain a lock for 200 milliseconds. If the lock could not be obtained within this timeout interval, the constructor returns. Afterward you can check whether or not the lock has been acquired, which can be done with an `if` statement on the `lock` variable because the `unique_lock` class defines a `bool` conversion operator. The timeout is specified using the `Chrono` library, discussed in Chapter 19.

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : mId(id), mNumIterations(numIterations)
    {
    }
    void operator()() const
    {
```

```

        for (int i = 0; i < mNumIterations; ++i) {
            unique_lock<timed_mutex> lock(mTimedMutex, 200ms);
            if (lock) {
                cout << "Counter " << mId << " has value ";
                cout << i << endl;
            } else {
                // Lock not acquired in 200 ms
            }
        }
    }
private:
    int mId;
    int mNumIterations;
    static timed_mutex mTimedMutex;
};

timed_mutex Counter::mTimedMutex;

```

If your compiler does not yet support the C++14 standard user-defined literals, then instead of 200ms, you have to write the following:

```
chrono::milliseconds(200)
```

Double-Checked Locking

You can use locks to implement the *double-checked locking pattern*.

WARNING *The double-checked locking pattern is explained here because you might encounter it in existing code. Double-checked locking is sensitive to race conditions, cache coherency, and so on; It is hard to get right. It's recommended to avoid this pattern as much as possible in new code. Instead, use other mechanisms such as simple locks, atomic variables, and call_once() without any double checking.*

Double-checked locking could for example, be used to make sure that a variable is initialized exactly once. The following example shows how you can implement this. It is called the double-checked locking algorithm because it is checking the value of the initialized variable twice, once before acquiring the lock and once right after acquiring the lock. The first initialized check is to prevent obtaining a lock when it is not needed and will increase performance. The second check is required to make sure that no other thread performed the initialization between the first initialized check and acquiring the lock:

```

void initializeSharedResources()
{
    // ... Initialize shared resources that will be used by multiple threads.
    cout << "Shared resources initialized." << endl;
}

atomic<bool> initialized(false);
mutex mut;

```

```

void func()
{
    if (!initialized) {
        unique_lock<mutex> lock(mut);
        if (!initialized) {
            initializeSharedResources();
            initialized = true;
        }
    }
    cout << "OK" << endl;
}
int main()
{
    vector<thread> threads;
    for (int i = 0; i < 5; ++i) {
        threads.push_back(thread{ func });
    }
    for (auto& t : threads) {
        t.join();
    }
}

```

The output clearly shows that only one thread has initialized the shared resources:

```

Shared resources initialized.
OK
OK
OK
OK
OK

```

NOTE For this example, it's recommended to use `call_once()` as demonstrated earlier in this chapter, instead of double-checked locking.

CONDITION VARIABLES

Condition variables allow a thread to block until a certain condition is set by another thread or until the system time reaches a specified time. They allow for explicit inter-thread communication. If you are familiar with multithreaded programming using the Win32 API, you can compare condition variables with *event objects* in Windows.

There are two kinds of condition variables available, both defined in the `<condition_variable>` header file:

- `std::condition_variable`: A condition variable that can wait only on a `unique_lock<mutex>`, which, according to the standard, allows for maximum efficiency on certain platforms.
- `std::condition_variable_any`: A condition variable that can wait on any kind of object, including custom lock types.

The `condition_variable` class supports the following methods.

- `notify_one();`
Wakes up one of the threads waiting on this condition variable. This is similar to an auto-reset event in Windows.
- `notify_all();`
Wakes up all threads waiting on this condition variable.
- `wait(unique_lock<mutex>& lk);`
The thread calling `wait()` should already have acquired a lock on `lk`. The effect of calling `wait()` is that it atomically calls `lk.unlock()` and then blocks the thread, waiting for a notification. When the thread is unblocked by a `notify_one()` or `notify_all()` call in another thread, the function calls `lk.lock()` again, possibly blocking on the lock and then returning.
- `wait_for(unique_lock<mutex>& lk, const chrono::duration<Rep, Period>& rel_time);`
Similar to the previous `wait()` method, except that the thread is unblocked by a `notify_one()` call, a `notify_all()` call, or when the given timeout has expired.
- `wait_until(unique_lock<mutex>& lk, const chrono::time_point<Clock, Duration>& abs_time);`
Similar to `wait()`, except that the thread is unblocked by a `notify_one()` call, a `notify_all()` call, or when the system time passes the given absolute time.

There are also versions of `wait()`, `wait_for()`, and `wait_until()` that accept an extra predicate parameter. For instance, the version of `wait()` accepting an extra predicate is equivalent to the following:

```
while (!predicate())
    wait(lk);
```

The `condition_variable_any` class supports the same methods as the `condition_variable` class except that it accepts any kind of lock class instead of only a `unique_lock<mutex>`. Your lock class should have a `lock()` and `unlock()` method.

Threads waiting on a condition variable can wake up when another thread calls `notify_one()` or `notify_all()`, or with a relative timeout, or when the system time reaches a certain time, but can also wake up *spuriously*. This means that a thread can wake up even if no other thread has called any notify method. Thus, when a thread waits on a condition variable and wakes up, it needs to check whether it woke up because of a notify or not. One way to check for this is using one of the versions of `wait()` accepting a predicate.

As an example, condition variables can be used for background threads processing items from a queue. You can define a queue in which you insert items to be processed. A background thread waits until there are items in the queue. When an item is inserted into the queue, the thread wakes up,

processes the item, and goes back to sleep, waiting for the next item. Suppose you have the following queue:

```
std::queue<std::string> mQueue;
```

You need to make sure only one thread is modifying this queue at any given time. You can do this with a mutex:

```
std::mutex mMutex;
```

To be able to notify a background thread when an item is added, you need a condition variable:

```
std::condition_variable mCondVar;
```

A thread that wants to add an item to the queue first acquires a lock on the mutex, adds the item to the queue, and notifies the background thread. You can call `notify_one()` or `notify_all()` whether you currently have the lock or not. Both will work.

```
// Lock mutex and add entry to the queue.
unique_lock<mutex> lock(mMutex);
mQueue.push(entry);
// Notify condition variable to wake up thread.
mCondVar.notify_all();
```

The background thread waits for notifications in an infinite loop, as follows. Note the use of `wait()` accepting a predicate to correctly handle spurious wake-ups. The predicate checks if there is something in the queue. When the call to `wait()` returns, you are sure there is something in the queue.

```
unique_lock<mutex> lock(mMutex);
while (true) {
    // Wait for a notification.
    mCondVar.wait(lock, []{ return !mQueue.empty(); });
    // Condition variable is notified, so something is in the queue.
    // Process queue item...
}
```

The section, “Example: Multithreaded Logger Class,” toward the end of this chapter provides a complete example of how to use condition variables to send notifications to other threads.

The standard also defines a helper function called `std::notify_all_at_thread_exit(cond, lk)` where `cond` is a condition variable and `lk` is a `unique_lock<mutex>` instance. A thread calling this function should already have acquired the lock `lk`. When the thread exits, it automatically executes the following:

```
lk.unlock();
cond.notify_all();
```

NOTE *The lock lk stays locked until the thread exits. So, you need to make sure that this does not cause any deadlocks in your code, for example due to wrong lock ordering. Deadlocks are discussed earlier in this chapter.*

FUTURES

As discussed earlier in this chapter, using `std::thread` to launch a thread that calculates a single result does not make it easy to get the computed result back once the thread has finished executing. Another problem with `std::thread` is handling errors like exceptions. If a thread throws an exception and this exception is not handled by the thread itself, the C++ runtime calls `std::terminate`, which usually terminates the whole application. You can avoid this by using `std::future`, which is able to transport an uncaught exception to another thread, which can then handle the exception however it wants. Of course, it's good practice to always try to handle exceptions in the threads themselves as much as possible, preventing them from leaving the thread.

`std::future` and `std::promise` work together to make it easier to retrieve a result from a function that ran in the same thread or in another thread. Once a function, running in the same thread or in another thread, has calculated the value that it wants to return, it puts this value in a `promise`. This value can then be retrieved through a `future`. You can think of a `future/promise` pair as an inter-thread communication channel for a result.

A thread that launches another thread to calculate a value can get this value as follows. `T` is the type of the calculated result:

```
future<T> fut = ...; // Is discussed later
T res = fut.get();
```

The call to `get()` retrieves the result and stores it in the variable `res`. If the other thread has not yet finished calculating the result, the call to `get()` blocks until the value becomes available. You can avoid blocking by first asking the `future` if there is a result available:

```
if (fut.wait_for(0)) { // Value is available
    T res = fut.get();
} else { // Value is not yet available
    ...
}
```

A `promise` is the input side for the result; `future` is the output side. A `promise` is something where a thread stores its calculated result. The following code demonstrates how a thread might do this:

```
promise prom = ...; // Is discussed later
T val = ...; // Calculate result value
prom.set_value(val);
```

If a thread encounters some kind of error during its calculation, it can store an exception in the `promise` instead of the value:

```
prom.set_exception(runtime_error("message"));
```

A thread that launches another thread to calculate something should give the `promise` to the newly launched thread, so that it can store the result in it. This is made easy with `std::packaged_task`, which automatically links a `future` and a `promise`. The following code demonstrates this feature. It creates a `packaged_task`, which executes the given lambda expression in a separate thread. The lambda expression accepts two arguments and returns the sum of them as the result. The `future` is

retrieved from the `packaged_task` by calling `get_future()`. The thread is started by the third line, and the last line uses the `get()` function to wait for and retrieve the result from the launched thread:

```
packaged_task<int(int, int)> task([](int i1, int i2) { return i1 + i2; });
auto fut = task.get_future(); // Get the future
task(2, 3); // Launch the task
int res = fut.get(); // Retrieve the result
```

NOTE *This code is just for demonstration purposes. It launches a separate thread and then calls `get()`, which blocks until the result is calculated. This sounds like a very expensive function call. In real-world applications you use the promise/future model by periodically checking if there is a result available in the future (using `wait_for()` as discussed earlier), or by using a synchronization mechanism such as a condition variable. When the result is not yet available, you can do something else in the meantime, instead of blocking.*

If you want to give the C++ runtime more control over whether or not a thread is created to calculate something, you can use `std::async()`. It accepts a function to be executed and returns a future that you can use to retrieve the result. There are two ways in which `async()` can call your function:

- Creating a new thread to run your function asynchronously
- Running your function at the time you call `get()` on the returned future

If you call `async()` without additional arguments, the runtime automatically chooses one of the two methods depending on factors like the number of processors in your system and the amount of concurrency already taking place. You can force the runtime to use one or the other method by specifying a `launch::async` (create a new thread) or `launch::deferred` (use current thread) policy argument. The following example demonstrates the use of `async()`:

```
int calculate()
{
    return 123;
}
int main()
{
    auto fut = async(calculate);
    //auto fut = async(launch::async, calculate);
    //auto fut = async(launch::deferred, calculate);
    // Do some more work...
    // Get result
    int res = fut.get();
    cout << res << endl;
}
```

As you can see in this example, `std::async()` is one of the easiest methods to perform some calculations in another thread or the same thread, and retrieve the result afterwards.

NOTE *A future returned by a call to `async()` blocks in its destructor until the result is available.*

Exception Handling

A big advantage of using futures is that they automatically transport exceptions between threads. At the time you call `get()` on a future, you receive the requested result, or, any exception that occurred in the thread is rethrown in the thread calling `get()` and you can catch them using a normal `try/catch` block. Here is an example:

```
int calculate()
{
    throw runtime_error("Exception thrown from a thread.");
}
int main()
{
    // Use launch::async policy to force a new thread.
    auto fut = async(launch::async, calculate);
    // Do some more work...
    // Get result
    try {
        int res = fut.get();
        cout << res << endl;
    } catch (const exception& ex) {
        cout << "Caught exception: " << ex.what() << endl;
    }
}
```

EXAMPLE: MULTITHREADED LOGGER CLASS

This section demonstrates how to use threads, the mutual exclusion and lock classes, and condition variables to write a multithreaded `Logger` class. The class allows log messages to be added to a queue from different threads. The `Logger` class itself processes this queue in another background thread that serially writes the log messages to a file. The class will be designed in two iterations to show you some examples of problems you will encounter when writing multithreaded code.

The C++ standard does not have a thread-safe queue, thus it is obvious that you have to protect access to the queue with a mutex to prevent multiple threads from reading/writing to the queue at the same time. Based on that, you might define the `Logger` class as follows:

```
class Logger
{
public:
    // Starts a background thread writing log entries to a file.
    Logger();
    // Prevent copy construction and assignment.
    Logger(const Logger& src) = delete;
    Logger& operator=(const Logger& rhs) = delete;
    // Add log entry to the queue.
    void log(const std::string& entry);
private:
    // The function running in the background thread.
    void processEntries();
    // Mutex and condition variable to protect access to the queue.
```

```

        std::mutex mMutex;
        std::condition_variable mCondVar;
        std::queue<std::string> mQueue;
        // The background thread.
        std::thread mThread;
    };
}

```

The implementation is as follows. Note that this initial design has a couple of problems and when you try to run it, it might behave strangely or even crash. This is discussed and solved in the next iteration of the `Logger` class. The inner `while` loop in the `processEntries()` method is also worth looking at. It processes all messages in the queue one at a time, and acquires and releases the lock on each iteration. This is done to make sure the loop doesn't keep the lock for too long, blocking other threads.

```

Logger::Logger()
{
    // Start background thread.
    mThread = thread{ &Logger::processEntries, this };
}
void Logger::log(const std::string& entry)
{
    // Lock mutex and add entry to the queue.
    unique_lock<mutex> lock(mMutex);
    mQueue.push(entry);
    // Notify condition variable to wake up thread.
    mCondVar.notify_all();
}
void Logger::processEntries()
{
    // Open log file.
    ofstream ofs("log.txt");
    if (ofs.fail()) {
        cerr << "Failed to open logfile." << endl;
        return;
    }
    // Start processing loop.
    unique_lock<mutex> lock(mMutex);
    while (true) {
        // Wait for a notification.
        mCondVar.wait(lock);
        // Condition variable is notified, so something might be in the queue.
        lock.unlock();
        while (true) {
            lock.lock();
            if (mQueue.empty()) {
                break;
            } else {
                ofs << mQueue.front() << endl;
                mQueue.pop();
            }
            lock.unlock();
        }
    }
}

```

This `Logger` class can be tested with the following test code. It launches a number of background threads, all logging a few messages to the same `Logger` instance:

```
void logSomeMessages(int id, Logger& logger)
{
    for (int i = 0; i < 10; ++i) {
        stringstream ss;
        ss << "Log entry " << i << " from thread " << id;
        logger.log(ss.str());
    }
}
int main()
{
    Logger logger;
    vector<thread> threads;
    // Create a few threads all working with the same Logger instance.
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(logSomeMessages, i, ref(logger));
    }
    // Wait for all threads to finish.
    for (auto& t : threads) {
        t.join();
    }
}
```

If you build and run this naïve initial version on a multicore machine, you will notice that the background `Logger` thread is terminated abruptly when the `main()` function finishes. This means that messages still in the queue are not written to the file on disk. Some runtime libraries even issue an error or generate a crash dump when the background `Logger` thread is abruptly terminated. You need to add a mechanism to gracefully shut down the background thread and wait until the background thread is completely shut down before terminating the application itself. This can be done by adding a destructor and an atomic Boolean member variable to the class. The new definition of the class is as follows:

```
class Logger
{
public:
    // Gracefully shut down background thread.
    virtual ~Logger();
    // Other public members omitted for brevity
private:
    // Boolean telling the background thread to terminate.
    std::atomic<bool> mExit;
    // Other members omitted for brevity
};
```

The `Logger` constructor needs to initialize `mExit`. The destructor sets it to `true`, wakes up the thread, and then waits until the thread is shut down. The destructor acquires a lock on `mMutex` before setting `mExit` to `true` and before calling `notify_all()`. This is to prevent a race condition and deadlock with `processEntries()`. `processEntries()` could be at the beginning of its `while` loop right after having checked `mExit` and right before the call to `wait()`. If the main thread calls the `Logger` destructor at that very moment, and the destructor wouldn't acquire a lock on `mMutex`, then the destructor sets `mExit` to `true` and calls `notify_all()` after `processEntries()` has checked `mExit` and before `processEntries()` is waiting on the condition variable, thus `processEntries()` will

not see the new value of `mExit` and it will miss the notification. In that case, the application is in a deadlock situation, because the destructor is waiting on the `join()` call and the background thread is waiting on the condition variable. Note that the destructor must release the lock on `mMutex` before calling `join()`, which explains the extra code block using curly brackets.

```
Logger::Logger() : mExit(false)
{
    // Start background thread.
    mThread = thread{ &Logger::processEntries, this };
}
Logger::~Logger()
{
{
    unique_lock<mutex> lock(mMutex);
    // Gracefully shut down the thread by setting mExit
    // to true and notifying the thread.
    mExit = true;
    // Notify condition variable to wake up thread.
    mCondVar.notify_all();
}
// Wait until thread is shut down. This should be outside the above code
// block because the lock on mMutex must be released before calling join()!
mThread.join();
}
```

The `processEntries()` method needs to check this Boolean variable and terminate the processing loop when it's true:

```
void Logger::processEntries()
{
    // Open log file.
    ofstream ofs("log.txt");
    if (ofs.fail()) {
        cerr << "Failed to open logfile." << endl;
        return;
    }
    // Start processing loop.
    unique_lock<mutex> lock(mMutex);
    while (true) {
        if (!mExit) { // Only wait for notifications if we don't have to exit.
            // Wait for a notification.
            mCondVar.wait(lock);
        }
        // Condition variable is notified, so something might be in the queue
        // and/or we need to shut down this thread.
        lock.unlock();
        while (true) {
            lock.lock();
            if (mQueue.empty()) {
                break;
            } else {
                ofs << mQueue.front() << endl;
                mQueue.pop();
            }
            lock.unlock();
        }
    }
}
```

```

        if (mExit) {
            break;
        }
    }
}

```

Note that you cannot just check for `mExit` in the condition for the outer `while` loop because even when `mExit` is `true`, there might still be log entries in the queue that need to be written.

You can add artificial delays on specific places in your multithreaded code to trigger certain behavior. Note that such delays should only be added for testing, and should be removed from your final code! For example, to test that the race-condition with the destructor is solved, you can remove any calls to `log()` from the main program causing it to almost immediately call the destructor of the `Logger` class, and add the following delay:

```

void Logger::processEntries()
{
    // Omitted for brevity

    // Start processing loop.
    unique_lock<mutex> lock(mMutex);
    while (!mExit) {

        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

        // Wait for a notification.
        mCondVar.wait(lock);

        // Omitted for brevity
    }
}

```

THREAD POOLS

Instead of creating and deleting threads dynamically throughout your program lifetime, you can create a pool of threads that can be used as needed. This technique is often used in programs that want to handle some kind of event in a thread. In most environments, the ideal number of threads is equal to the number of processing cores. If there are more threads than cores, threads will have to be suspended to allow other threads to run, and this will ultimately add overhead. Note that while the ideal number of threads is equal to the number of cores, this applies only in the case where the threads are compute bound and cannot block for any other reason, including I/O. When threads can block, it is often appropriate to run more threads than there are cores. Determining the optimal number of threads in such cases may involve throughput measurements.

Because not all processing is identical, it is not uncommon to have threads from a thread pool receive, as part of their input, a function object or lambda expression that represents the computation to be done.

Because threads from a thread pool are pre-existing, it is vastly more efficient for the operating system to schedule one to run than it is to create one in response to an input. Furthermore, the use of a thread pool allows you to manage the number of threads created, so depending on the platform, you may have just one thread or thousands of threads.

There are several libraries available that implement thread pools; for example, Intel Threading Building Blocks (TBB), Microsoft Parallel Patterns Library (PPL), and so on. It's recommended to use such a library for your thread pools instead of writing your own implementation. If you do want to implement a thread pool yourself, it can be done in a similar way as an object pool. Chapter 25 gives an example implementation of an object pool.

THREADING DESIGN AND BEST PRACTICES

This section briefly lists a couple of best practices related to multithreaded programming.

- **Before terminating the application, always use `join()` to wait for background threads to finish:** Make sure you use `join()` on all background threads before terminating your application. This will make sure all those background threads have the time to do proper cleanup. Background threads for which there is no `join()` will terminate abruptly when the main thread is terminated.
- **The best synchronization is no synchronization:** Multithreaded programming becomes much easier if you manage to design your different threads in such a way that all threads working on shared data read only from that shared data and never write to it, or only write to parts never read by other threads. In that case there is no need for any synchronization and you cannot have problems like race conditions or deadlocks.
- **Try to use the single-thread ownership pattern:** This means that a block of data is owned by no more than one thread at a time. Owning the data means that no other thread is allowed to read/write to the data. When the thread is finished with the data, the data can be passed off to another thread, which now has sole and complete responsibility/ownership of the data. No synchronization is necessary in this case.
- **Use atomic types and operations when possible:** Atomic types and atomic operations make it easier to write race-condition and deadlock-free code, because they handle synchronization automatically. If atomic types and operations are not possible in your multithreaded design, and you need shared data, you have to use a mutual exclusion mechanism to ensure proper synchronization.
- **Use locks to protect mutable shared data:** If you need mutable shared data to which multiple threads can write, and you cannot use atomic types and operations; you have to use a locking mechanism to make sure reads and writes between different threads are synchronized.
- **Release locks as soon as possible:** When you need to protect your shared data with a lock, make sure you release the lock as soon as possible. While a thread is holding a lock, it is blocking other threads waiting for the same lock, possibly hurting performance.
- **Make sure to acquire multiple locks in the same order:** If multiple threads need to acquire multiple locks, they must be acquired in the same order in all threads to prevent deadlocks. You should use the generic `std::lock()` or `std::try_lock()` to minimize the chance of violating lock ordering restrictions.

- **Use a multithreading-aware profiler:** Use a multithreading-aware profiler to find performance bottlenecks in your multithreaded applications and to find out if your multiple threads are indeed utilizing all available processing power in your system. An example of a multithreading-aware profiler is the profiler in certain editions of Microsoft Visual Studio.
- **Understand the multithreading support features of your debugger:** Most debuggers have at least basic support for debugging multithreaded applications. You should be able to get a list of all running threads in your application, and you should be able to switch to any one of those threads to inspect their call stack. You can use this, for example, to inspect deadlocks because you can see exactly what each thread is doing.
- **Use thread pools instead of creating and destroying a lot of threads dynamically:** Your performance decreases if you dynamically create and destroy a lot of threads. In that case it's better to use a thread pool to reuse existing threads.
- **Use higher-level multithreading libraries:** Where possible, use higher-level multithreading libraries such as Intel Threading Building Blocks (TBB), Microsoft Parallel Patterns Library (PPL), and so on, rather than reinventing the wheel. Multithreaded programming is hard to get right and is error prone. More often than not, your wheel may not be as round as you think.

SUMMARY

This chapter gave a brief overview of multithreaded programming using the standard C++ threading library. It explained how you can use atomic types and atomic operations to operate on shared data without having to use explicit locks. In case you cannot use these atomic types and operations, you learned how to use mutual exclusion mechanisms to ensure proper synchronization between different threads that need read/write access to shared data. You also saw how promises and futures represent a simple inter-thread communication channel; you can use futures to more easily get a result from a background thread. The chapter finished with a number of best practices for multithreaded application design.

As mentioned in the introduction, this chapter tried to touch on all the functionality provided by the standard C++ threading library, but due to space constraints, it cannot go into all the details of multithreaded programming. There are books available that discuss nothing but multithreading. See Appendix B for a few references.

PART V

C++ Software Engineering

- **CHAPTER 24:** Maximizing Software Engineering Methods
- **CHAPTER 25:** Writing Efficient C++
- **CHAPTER 26:** Conquering Debugging

24

Maximizing Software Engineering Methods

WHAT'S IN THIS CHAPTER?

- What a software life cycle model is, with examples of the Stagewise Model, the Waterfall Model, the Spiral Model, and RUP
- What software engineering methodologies are, with examples of Agile, Scrum, XP, and Software Triage
- What Source Code Control means

Chapter 24 starts the last part of this book, which is about *software engineering*. This part describes software engineering methods, code efficiency, and software debugging.

When you first learned how to program, you were probably on your own schedule. You were free to do everything at the last minute if you wanted to, and you could radically change your design during implementation. When coding in the professional world, however, programmers rarely have such flexibility. Even the most liberal engineering managers admit that some amount of process is necessary. Knowing the software engineering process is as important these days as knowing how to code.

This chapter surveys various approaches to software engineering. It does not go into great depth on any one approach — there are plenty of excellent books on software engineering processes. The idea is to cover some different types of processes in broad strokes so you can compare and contrast them. I try not to advocate or discourage any particular methodology. Rather, I hope that by learning about the tradeoffs of several different approaches, you'll be able to construct a process that works for you and the rest of your team. Whether you're a contractor working alone on projects, or your team consists of hundreds of engineers on several continents, understanding different approaches to software development will help your job on a daily basis.

The last part of this chapter discusses Source Code Control solutions which make it easier to manage source code and keep track of its history. A Source Code Control solution is mandatory in every company to avoid a source code maintenance nightmare.

THE NEED FOR PROCESS

The history of software development is filled with tales of failed projects. From over-budget and poorly marketed consumer applications to grandiose mega-hyped operating systems, it seems that no area of software development is free from this trend.

Even when software successfully reaches users, bugs have become so commonplace that end users are forced to endure constant updates and patches. Sometimes the software does not accomplish the tasks it is supposed to or doesn't work the way the user would expect. These issues all point to a common truism of software — writing software is hard.

One wonders why software engineering seems to differ from other forms of engineering in its frequency of failures. While cars do have their share of bugs, you rarely see them stop suddenly and demand a reboot due to a buffer overflow (though as more car components become software-driven, you just may.) Your TV may not be perfect, but you don't have to upgrade to version 2.3 to get Channel 6 to work.

Is it the case that other engineering disciplines are just more advanced than software? Is a civil engineer able to construct a working bridge by drawing upon the long history of bridge building? Are chemical engineers able to build a compound successfully because most of the bugs were worked out in earlier generations?

NOTE *Is software too new, or is it really a different type of discipline with inherent qualities contributing to the occurrence of bugs, unusable results, and doomed projects?*

It certainly seems as if there's something different about software. For one thing, technology changes rapidly in software, creating uncertainty in the software development process. Even if an earth-shattering breakthrough does not occur during your project, the pace of the industry leads to problems. Software often needs to be developed quickly because competition is fierce.

Software development schedules can also be unpredictable. Accurate scheduling is nearly impossible when a single gnarly bug can take days or even weeks to fix. Even when things seem to be going according to schedule, the widespread tendency of product definition changes (*feature creep*) can throw a wrench in the process.

Software is complex. There is no easy and accurate way to prove that a program is bug-free. Buggy or messy code can have an impact on software for years if it is maintained through several versions. Software systems are often so complex that when staff turnover occurs, nobody wants to get anywhere near the messy code that forgotten engineers have left behind. This leads to a cycle of endless patching, hacks, and workarounds.

Of course, standard business risks apply to software as well. Marketing pressures and miscommunication get in the way. Many programmers try to steer clear of corporate politics, but it's not uncommon to have adversity between the development and product marketing groups.

All of these factors working against software engineering products indicate the need for some sort of process. Software projects are big, complicated, and fast-paced. To avoid failure, engineering groups need to adopt a system to control this unwieldy process.

SOFTWARE LIFE CYCLE MODELS

Complexity in software isn't new. The need for a formalized process was recognized decades ago. Several approaches to modeling the *software life cycle* have attempted to bring some order to the chaos of software development by defining the software process in terms of steps from the initial idea to the final product. These models, refined over the years, guide much of software development today.

The Stagewise Model and Waterfall Model

A classic life cycle model for software is the *Stagewise Model*. This model is based on the idea that software can be built almost like following a recipe. There is a set of steps that, if followed correctly, will yield a mighty fine chocolate cake, or program as the case may be. Each stage must be completed before the next stage can begin, as shown in Figure 24-1.

The process starts with formal planning, including gathering an exhaustive list of requirements. This list defines feature completeness for the product. The more specific the requirements are, the more likely that the project will succeed. Next, the software is designed and fully specified. The design step, like the requirements step, needs to be as specific as possible to maximize the chance of success. All design decisions are made at this time, often including pseudocode and the definition of specific subsystems that will need to be written. Subsystem owners work out how their code will interact, and the team agrees on the specifics of the architecture. Implementation of the design occurs next. Because the design has been fully specified, the code needs to adhere strongly to the design or else the pieces won't fit together. The final four stages are reserved for unit testing, subsystem testing, integration testing, and evaluation.

The main problem with the Stagewise Model is that, in practice, it is nearly impossible to complete one stage without at least exploring the next stage. A design cannot be set in stone without at least writing *some* code. Furthermore, what is the point of testing if the model doesn't provide a way to go back to the coding phase?

A number of refinements to the Stagewise Model were formalized as the *Waterfall Model* in the early 1970s. The main advancement that the Waterfall Model brought was a notion of feedback between stages. While it still stresses a rigorous process of planning, designing, coding, and testing, successive stages can overlap in part. Figure 24-2 shows an example of the Waterfall Model,

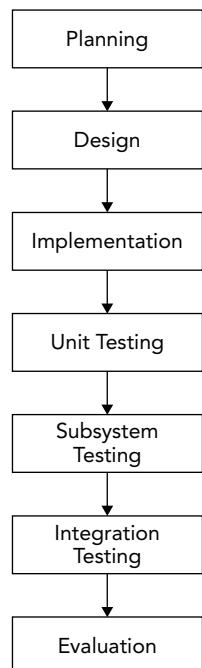


FIGURE 24-1

illustrating the feedback and overlapping refinements. Feedback allows lessons learned in one phase to result in changes to the previous phase. Overlap permits activity in two phases to occur simultaneously.

Various incarnations of the Waterfall Model have refined the process in different ways. For example, some plans include a “feasibility” step where experiments are performed before formal requirements are even gathered.

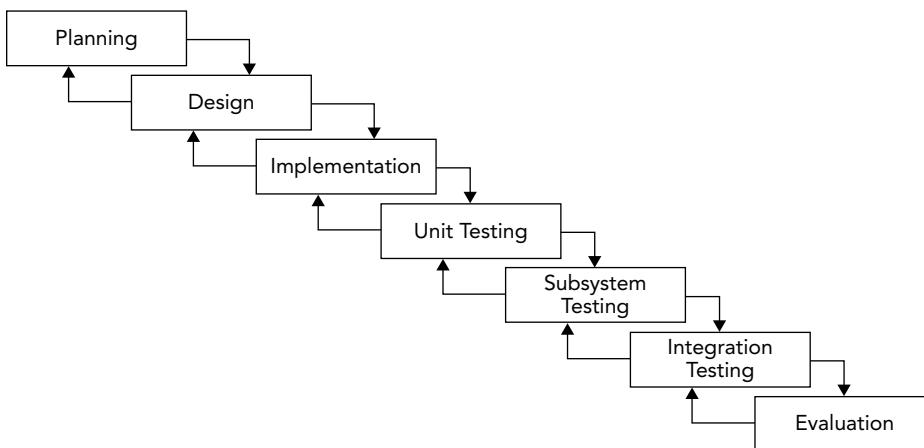


FIGURE 24-2

Benefits of the Waterfall Model

The value of the Waterfall Model lies in its simplicity. You, or your manager, may have followed this approach in past projects without formalizing it or recognizing it by name. The underlying assumption behind the Stagewise Model and Waterfall Model is that as long as each step is accomplished as completely and accurately as possible, subsequent steps will go smoothly. As long as all of the requirements are carefully specified in the first step, and all the design decisions and problems are hashed out in the second step, implementation in the third step should be a simple matter of translating the designs into code.

The simplicity of the Waterfall Model makes project plans based on this system organized and easy to manage. Every project is started the same way: by exhaustively listing all the features that are necessary. Managers using this approach can require that by the end of the design phase, for example, all engineers in charge of a subsystem submit their design as a formal design document or a functional subsystem specification. The benefit for the manager is that by having engineers specify requirements and design upfront, risks are, hopefully, minimized.

From the engineer’s point of view, the Waterfall Model forces resolution of major issues upfront. All engineers will need to understand their project and design their subsystem before writing a significant amount of code. Ideally, this means that code can be written once instead of hacked together or rewritten when the pieces don’t fit.

For small projects with very specific requirements, the Waterfall Model can work quite well. Particularly for consulting arrangements, it has the advantage of specifying specific metrics for success at the start of the project. Formalizing requirements helps the consultant to produce exactly what the client wants and forces the client to be specific about the goals for the project.

Drawbacks of the Waterfall Model

In many organizations, and almost all modern software engineering texts, the Waterfall Model has fallen out of favor. Critics disparage its fundamental premise that software development tasks happen in discrete linear steps. While the Waterfall Model allows for the overlapping of phases, it does not allow backward movement to a large degree. In many projects today, requirements come in throughout the development of the product. Often, a potential customer will request a feature that is necessary for the sale, or a competitor's product will have a new feature that requires parity.

NOTE *The upfront specification of all requirements makes the Waterfall Model unusable for many organizations because it is not dynamic enough.*

Another drawback is that in an effort to minimize risk by making decisions as formally and early as possible, the Waterfall Model may actually be hiding risk. For example, a major design issue might be undiscovered, glossed over, forgotten, or purposely avoided in the design phase. By the time integration testing reveals the mismatch, it may be too late to save the project. A major design flaw has arisen but, according to the Waterfall Model, the product is one step away from shipping! A mistake anywhere in the waterfall process will likely lead to failure at the end of the process. Early detection is difficult and rare.

If you use the Waterfall Model, it is often necessary to make it more flexible by taking cues from other approaches.

The Spiral Model

The *Spiral Model* was proposed by Barry W. Boehm in 1986 as a risk-driven software development process. There are other spiral-like models. The one discussed in this section is part of a family of techniques known as *iterative processes*. The fundamental idea is that it's okay if something goes wrong because you'll fix it the next time around. A single spin through this Spiral Model is shown in Figure 24-3.

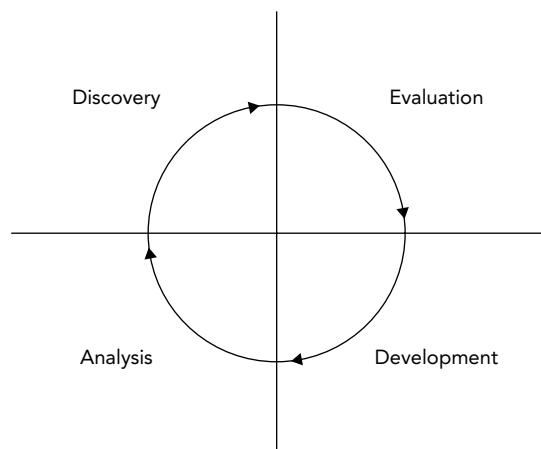


FIGURE 24-3

The phases of the Spiral Model are similar to the steps of the Waterfall Model. The discovery phase involves discovering requirements, determining objectives, determining alternatives (design alternatives, reuse, buying third party libraries, and so on), as well as determining any constraints. During the evaluation phase, implementation alternatives are evaluated, risks are analysed, and prototype options are considered. In the Spiral Model, particular attention is paid to evaluating and resolving risks in the evaluation phase. The tasks deemed most risky are the ones that are implemented in the current cycle of the spiral. The tasks in the development phase are determined by the risks identified in the evaluation phase. For example, if evaluation reveals a risky algorithm that may be impossible to implement, the main task for development in the current cycle will be modeling, building, and testing that algorithm. The fourth phase is reserved for analysis and planning. Based on the results of the current cycle, the plan for the subsequent cycle is formed.

Figure 24-4 shows an example of three cycles through the spiral in the development of an operating system. The first cycle yields a plan containing the major requirements for the product. The second cycle results in a prototype showing the user experience. The third cycle builds a component that is determined to be a high risk.

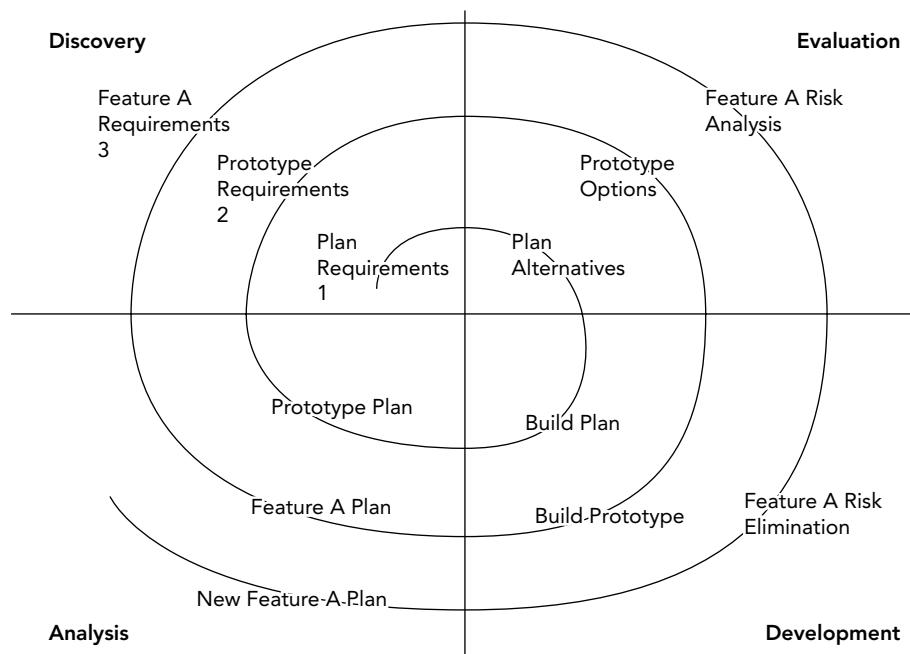


FIGURE 24-4

Benefits of the Spiral Model

The Spiral Model can be viewed as the application of an iterative approach to the best that the Waterfall Model has to offer. Figure 24-5 shows the Spiral Model as a waterfall process that has been modified to allow iteration. Hidden risks and a linear development path, the main drawbacks of the Waterfall Model, are resolved through iterative cycles.

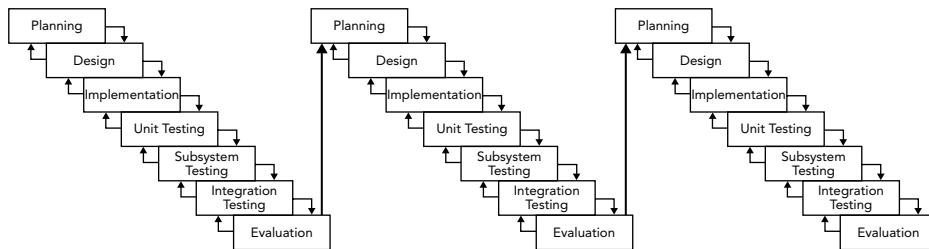


FIGURE 24-5

Performing the riskiest tasks first is another benefit. By bringing risk to the forefront and acknowledging that new conditions can arise at any time, the Spiral Model avoids the hidden time bombs that can occur in the Waterfall Model. When unexpected problems arise, they can be dealt with by using the same four-stage approach that works for the rest of the process.

This iterative approach also allows for incorporating feedback from testers. For example, an early version of the product can be released for internal or even external evaluation. These testers could, for example, say that a certain feature is missing, or an existing feature is not working as expected. The Spiral Model has a built-in mechanism to react to such input.

Finally, by repeatedly analyzing after each cycle and building new designs, the practical difficulties with the design-then-implement approach are virtually eliminated. With each cycle, there is more knowledge of the system that can influence the design.

Drawbacks of the Spiral Model

The main drawback of the Spiral Model is that it can be difficult to scope each iteration small enough to gain real benefit. In a worst-case scenario, the Spiral Model can degenerate into the Waterfall Model because the iterations are too long. Unfortunately, the Spiral Model only *models* the software life cycle. It cannot prescribe a specific way to break down a project into single-cycle iterations because that division varies from project to project.

Other possible drawbacks are the overhead of repeating all four phases for each cycle and the difficulty of coordinating cycles. Logistically, it may be difficult to assemble all the group members for design discussions at the right time. If different teams are working on different parts of the product simultaneously, they are probably operating in parallel cycles, which can get out of sync. For example, the user interface group could be ready to start the discovery phase of the Window Manager cycle, but the core OS group could still be in the development phase of the memory subsystem.

The Rational Unified Process

The *Rational Unified Process (RUP)* is one of the best known refinements of the unified process. It is a disciplined and formal approach to managing the software development process. The most important characteristic of the RUP is that, unlike the Spiral Model or the Waterfall Model, RUP is more than just a theoretical process model. RUP is actually a software product, marketed and sold by Rational Software, a division of IBM. Treating the process as software is compelling for a number of reasons:

- The process itself can be updated and refined, just as software products periodically have updates.

- Rather than simply suggesting a development framework, RUP includes a set of software tools for working with that framework.
- As a product, RUP can be rolled out to the entire engineering team so that all members are using the exact same processes and tools.
- Like many software products, RUP can be customized to the needs of the users.

RUP as a Product

As a product, the RUP takes the form of a suite of software applications that guides developers through the software development process. The product also offers specific guidance for other Rational products, such as the Rational Rose visual modeling tool and the Rational ClearCase configuration management tool. Extensive groupware communication tools are included as part of the “marketplace of ideas” that allow developers to share knowledge.

One of the basic principles behind RUP is that each iteration on a development cycle should have a tangible result. During the Rational Unified Process, users will create numerous designs, requirement documents, reports, and plans. The RUP software provides visualization and development tools for the creation of these artifacts.

RUP as a Process

Defining an accurate model is the central principle of RUP. Models, according to RUP, help explain the complicated structures and relationships in the software development process. In RUP, models are often expressed in Unified Modeling Language (UML) format.

RUP defines each part of the process as an individual *workflow*. Workflows represent each step of a process in terms of who is responsible for it, what tasks are being performed, the artifacts or results of these tasks, and the sequence of events that drives the tasks. Almost everything about RUP is customizable, but several *core process workflows* are defined “out of the box” by RUP.

The core process workflows bear some resemblance to the stages of the Waterfall Model, but each one is iterative and more specific in definition. The *business modeling workflow* models business processes, usually with the goal of driving software requirements forward. The *requirements workflow* creates the requirements definition by analyzing the problems in the system and iterating on its assumptions. The *analysis and design workflow* deals with system architecture and subsystem design. The *implementation workflow* covers the modeling, coding, and integration of software subsystems. The *test workflow* models the planning, implementation, and evaluation of software quality tests. The *deployment workflow* is a high-level view on overall planning, releasing, supporting, and testing workflows. The *configuration management workflow* goes from new project conception to iteration and end-of-product scenarios. Finally, the *environment workflow* supports the engineering organization through the creation and maintenance of development tools.

RUP in Practice

RUP is aimed mainly at larger organizations and offers several advantages over the adoption of traditional life cycle models. Once the team has gotten over the learning curve to use the software, all members will be using a common platform for designing, communicating, and implementing their

ideas. The process can be customized to the needs of the team, and each stage reveals a wealth of valuable artifacts that document each phase of the development.

A product like RUP can be too heavyweight for some organizations. Teams with diverse development environments or tight engineering budgets might not want to or be able to standardize on a software-based development system. The learning curve can also be a factor — new engineers that aren't familiar with the process software will have to learn how to use it while getting up to speed on the product and the existing code base.

SOFTWARE ENGINEERING METHODOLOGIES

Software life cycle models provide a formal way of answering the question “What do we do next?” but are rarely (with the exception of formalized systems like RUP) able to contribute an answer to the logical follow-up question, “How do we do it?” To provide some answers to the “how” question, a number of methodologies have been developed that provide practical rules of thumb for professional software development. Books and articles on software methodologies abound, but a few innovations deserve particular attention: *Agile*, *Scrum*, *Extreme Programming*, and *Software Triage*.

Agile

To address the shortcomings of the Waterfall Model, the *Agile Methodology* was introduced in 2001 in the form of an *Agile Manifesto*. The entire manifesto reads as follows (<http://agilemanifesto.org/>):

MANIFESTO FOR AGILE SOFTWARE DEVELOPMENT

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

As can be understood from this manifesto, the term *Agile* is only a high-level description. Basically, it tells you to make the process flexible so that customers' changes can easily be incorporated into the project during development.

Scrum

Agile is just a high-level foundation; it does not specify exactly how this should be implemented in real life. That's where *Scrum* comes into play; it's an Agile methodology with precise descriptions on how to use it on a daily base.

Just like the Spiral Model, Scrum is an iterative process. It is very popular as a means to manage software development projects. In Scrum, each iteration is called a *sprint cycle*. The sprint cycle is the central part of the Scrum process. The length of sprints, which should be decided at the beginning of the project, is typically between two and four weeks. At the end of each sprint, the goal is to have a version of the software available that is fully working and tested, and which represents a subset of the customers' requirements. Scrum recognizes that customers will often change their minds during the development and thus allows the result of each sprint to be shipped to the customer. This gives customers the opportunity to see iterative versions of the software and allows them to give feedback to the development team about potential issues.

Roles

The *Product Owner* (PO) is the connection to the customer and to other people. The PO writes high-level *user stories* based on input from the customer, gives each user story a priority, and puts the stories in the Scrum product backlog. Actually, everyone on the team is allowed to write high-level user stories for the product backlog, but the Product Owner decides which user stories are kept and which are removed.

The *Scrum Master* (SM) is responsible for keeping the process running and can be part of the team, although not the team leader, because with Scrum the team leads itself. The SM is the contact person for the team so that the rest of the team members can concentrate on their tasks. The SM ensures that the Scrum process is followed correctly by the team; for example, by organizing the Daily Scrum meetings, which are discussed later. The Scrum Master and Product Owner should be two different persons.

The third and final role in the Scrum process is the *Team* itself. Teams, which develop the software, should be kept small, preferably fewer than 10 members.

The Process

The Scrum process enforces a daily meeting called the *Daily Scrum* or *Standup*. In this meeting, all team members stand together with the Scrum Master. According to the Scrum process, this meeting should start every day at exactly the same time and location, and should be no longer than 15 minutes. During this meeting, all team members get to answer three questions:

- What did you do since the last Daily Scrum?
- What are you planning to do after the current Daily Scrum?
- What problems are you facing to reach your goal?

Problems faced by team members should be noted by the Scrum Master who will try to solve them after the Daily Scrum meeting.

Before the start of each sprint cycle there is a *Sprint Planning* meeting in which team members must decide which product features they will implement in the new sprint. This is formalized in a sprint

backlog. The features are selected from a *product backlog* containing prioritized user stories, which are high-level requirements of new features. User stories from the product backlog are broken down into smaller tasks with an *effort estimation* and are put in the sprint backlog. Once a sprint is in progress, the sprint backlog is frozen and cannot be changed during that sprint. The Sprint Planning should not take more than eight hours and is usually split into two parts: meeting with the Product Owner and the team to discuss the priority of product backlog items; and meeting with the team only to complete the sprint backlog.

In a Scrum team you will sometimes find a physical board with three columns: *To Do*, *In Progress*, and *Done*. Every task for the sprint is written on a small paper and stuck on the board in the correct column. Tasks are not assigned to people during a meeting; instead, every team member can go to the board, pick one of the *To Do* tasks, and move that paper to the *In Progress* column. When the team member is finished with that task, the paper is moved to the *Done* column. This method makes it easy for team members to quickly get an overview of the work that still needs to be done and what tasks are in progress or finished. Instead of a physical Scrum board, you can use a software solution to work with a virtual Scrum board.

Usually, a burn-down chart is also created every day that displays the days of the sprint on the horizontal axis and the remaining development hours on the vertical axis. This gives a quick overview of the progress made and can be used to determine whether all planned tasks can be completed during the sprint.

Once a sprint cycle is finished, there are two meetings: the *Sprint Review* and the *Sprint Retrospective*. The Sprint Review should be less than four hours and should discuss the results of the sprint cycle, including what tasks were completed and what tasks were not completed and why. This meeting also includes a *Demo* to demonstrate the implemented features. The Sprint Retrospective should have a three-hour maximum and should allow the team to think about how the last sprint cycle was executed. For example, the team can identify shortcomings in the process and adapt the process for the next sprint.

Benefits of Scrum

Scrum is resilient to unforeseen problems that come up during the development. When a problem pops up, it can be handled in one of the following sprints. The team is involved in every step of the project. It discusses user stories from the product backlog with the Product Owner and converts user stories into smaller tasks for inclusion in a sprint backlog. The team autonomously assigns work to its members with the aid of the Scrum tasks board. This board makes it easy to quickly see which team member is working on which task. And finally, the Daily Scrum meeting makes sure that everyone knows what is happening.

A huge benefit to the paying customer is a *Demo* that follows each sprint, which demonstrates the new iterative version of the project. The customer gets a quick sense of how the development is progressing and can make changes to requirements, which usually can be incorporated into a future sprint.

Drawbacks of Scrum

Some companies might find it difficult to accept that the team itself decides who is doing what. Tasks are not assigned to team members by a manager or a team leader. All members pick their own tasks from the Scrum tasks board.

The Scrum Master is a key person to make sure the team stays on track. It is very important that the SM trusts the team. Having too tight control over the team members will cause the Scrum process to fail.

A possible problem with Scrum is called *feature creep*. Scrum allows new user stories to be added to the product backlog during the development. The danger exists that project managers keep adding new features to the product backlog. This problem is best solved by deciding on a final release date, or the end date of the last sprint.

Extreme Programming (XP)

When a friend of mine arrived home from work a few years ago and told his wife that his company had adopted some of the principles of Extreme Programming, she joked, “I hope you wear a safety harness for that.” Despite the somewhat hokey name, Extreme Programming effectively bundles up the best of existing software development guidelines and new material into an increasingly popular methodology.

XP, popularized by Kent Beck in *eXtreme Programming eXplained* (Addison-Wesley, 1999), claims to take the best practices of good software development and turn them up a notch. For example, most programmers would agree that testing is a good thing. In XP, testing is deemed so good that you’re supposed to write the tests before you write the code.

XP in Theory

The Extreme Programming methodology is made up of 12 main guiding principles. These principles are manifested throughout all phases of the software development process and have a direct impact on the daily tasks of engineers.

Plan as You Go

In the Waterfall Model, planning happened once, at the beginning of the process. Under the Spiral Model, planning was the first phase of each iteration. In RUP, planning is an integral step in most of the workflows. Under XP, planning is more than just a step — it’s a never-ending task. XP teams start with a rough plan that captures the major points of the product being developed. Throughout the development process, the plan is refined and modified as necessary. The theory is that conditions are constantly changing and new information is obtained all the time.

Under XP, estimates for a given feature are always made by the person who will be implementing that particular feature. This helps to avoid situations where the implementer is forced to adhere to an unrealistic and artificial schedule. Initially, estimates are very rough, perhaps on the order of weeks for a feature. As the time horizon shortens, the estimates get more granular. Features are broken down into tasks taking no more than five days.

Build Small Releases

One of the theories of XP is that software projects grow risky and unwieldy when they try to accomplish too much at one time. Instead of massive software releases that involve core changes and several pages of release notes, XP advocates smaller releases with a timeframe closer to two months than 18 months. With such a short release cycle, only the most important features can make it into the product. This forces engineering and marketing to agree on what features are truly important.

Share a Common Metaphor

XP uses the term *metaphor* as other methodologies might use *architecture*. The idea is that all members of the team should share a common high-level view of the system. This isn't necessarily the specifics of how objects will communicate, or the exact APIs that will be written. Rather, the metaphor is the mental model for the components of the system. Team members should use the metaphor to drive shared terminology when discussing the project.

Simplify Your Designs

A mantra frequently sung by XP-savvy engineers is “avoid speculative generality.” This goes against the natural inclinations of many programmers. If you are given the task of designing a file-based object store, you may start down the path of creating the be-all, end-all solution to all file-based storage problems. Your design might quickly evolve to cover multiple languages and any type of object. XP says you should lean towards the other end of the generality continuum. Instead of making the ideal object store that will win awards and be celebrated by your peers, design the simplest possible object store that gets the job done. You should understand the current requirements and write your code to those specifications to avoid overly complex code.

It may be hard to get used to simplicity in design. Depending on the type of work you do, your code may need to exist for years and be used by other parts of the code that you haven't even dreamed of. As discussed in Chapter 6, the problem with building in functionality that *may* be useful in the future is that you don't know what those hypothetical use cases are, and there is no way to craft a good design that is purely speculative. Instead, XP says you should build something that is useful today and leave open the opportunity to modify it later.

Test Constantly

According to *eXtreme Programming eXplained*, “Any program feature without an automated test simply doesn't exist.” Extreme Programming is zealous about testing. Part of your responsibility as an XP engineer is to write the unit tests that accompany your code. A unit test is generally a small piece of code that makes sure that an individual piece of functionality works. For example, individual unit tests for a file-based object store may include `testSaveObject`, `testLoadObject`, and `testDeleteObject`.

XP takes unit testing one step further by suggesting that unit tests should be written before the actual code is written. Of course, the tests won't pass because the code hasn't been written yet. In theory, if your tests are thorough, you should know when your code is done because all the tests will complete successfully. I told you it was “extreme.”

Refactor When Necessary

Most programmers *refactor* their code from time to time. Refactoring is the process of redesigning existing working code to take into account new knowledge or alternate uses that have been discovered since the code was written. Refactoring is difficult to build into a traditional software engineering schedule because its results are not as tangible as implementing a new feature. Good managers, however, recognize its importance for long-term code maintainability.

The extreme way of refactoring is to recognize situations during development when refactoring is useful and to do the refactoring at that time. Instead of deciding at the start of a release which

existing parts of the product need design work, XP programmers learn to recognize the signs of code that is ready to be refactored. While this practice will almost certainly result in unexpected and unscheduled tasks, restructuring the code when appropriate should make future development easier.

Code in Pairs

XP suggests that all production code should be written by two people working side-by-side simultaneously, called *pair programming*. Obviously, only one person can actually be in control of the keyboard. The other person takes a high-level approach, thinking about issues such as testing, necessary refactoring, and the overall model of the project.

As an example, if you are in charge of writing the user interface for a particular feature of your application, you might want to ask the original author of the feature to sit down with you. She can advise you about the correct use of the feature, warn you about any “gotchas” you should watch out for, and help oversee your efforts at a high level. Even if you can’t acquire the help of the original author, just grabbing another member of the team can help. The theory is that working in pairs builds shared knowledge, ensures proper design, and puts an informal system of checks and balances in place.

Share the Code

In many traditional development environments, code ownership is strongly defined and often enforced. A friend of mine worked previously in an environment where the manager explicitly forbade checking in changes to code written by any other member of the team. XP takes the extreme opposite approach by declaring that the code is collectively owned by everybody.

Collective ownership is practical for a number of reasons. From a management point of view, it is less detrimental when a single engineer leaves suddenly because there are others who understand that part of the code. From an engineer’s point of view, collective ownership builds a common view of how the system works. This helps design tasks and frees the individual programmer to make any change that will add value to the overall project.

One important note about collective ownership is that it is not necessary for every programmer to be familiar with every single line of code. It is more of a mindset that the project is a team effort, and there is no reason for any one person to hoard knowledge.

Integrate Continuously

All programmers are familiar with the dreaded chore of integrating code. This is the task when you discover that your view of the object store is a complete mismatch with the way it was actually written. When subsystems come together, problems are exposed. XP recognizes this phenomenon and advocates integrating code into the project frequently as it is being developed.

XP suggests a specific method for integration. Two programmers (the pair that developed the code) sit down at a designated “integration station” and merge the code in together. The code is not checked in until it passes 100 percent of the tests. By having a single station, conflicts are avoided and integration is clearly defined as a step that must occur before a check-in.

A similar approach can still work on an individual level. Engineers run tests individually or in pairs before checking code into the repository. A designated machine continually runs automated tests.

When the automated tests fail, the team receives an email indicating the problem and listing the most recent check-ins.

Work Sane Hours

XP has a thing or two to say about the hours you've been putting in. The claim is that a well-rested programmer is a happy and productive programmer. XP advocates a work week of approximately 40 hours and warns against putting in overtime for more than two consecutive weeks.

Of course, different people need different amounts of rest. The main idea, though, is that if you sit down to write code without a clear head, you're going to write poor code and abandon many of the XP principles.

Have a Customer on Site

Since an XP-savvy engineering group constantly refines its product plan and builds only what is currently necessary, having a customer contribute to the process is very valuable. Although it is not always possible to convince a customer to be physically present during development, the idea that there should be communication between engineering and the end user is clearly a valuable notion. In addition to assisting with the design of individual features, customers can help prioritize tasks by conveying their individual needs.

Share Common Coding Standards

Due to the collective ownership guideline and the practice of pair programming, coding in an extreme environment can be difficult if each engineer has her own naming and indenting conventions. XP doesn't advocate any particular style, but supplies the guideline that if you can look at a piece of code and immediately identify the author, your group probably needs better definition of its coding standards.

For additional information on various approaches to coding style, see Chapter 3.

XP in Practice

XP purists claim that the 12 tenets of Extreme Programming are so intertwined that adopting some of them without others would largely ruin the methodology. For example, pair programming is vital to testing because if you can't determine how to test a particular piece of code, your partner can help. Also, if you're tired one day and decide to skip the testing, your partner will be there to evoke feelings of guilt.

Some of the XP guidelines, however, can prove difficult to implement. To some engineers, the idea of writing tests before code is too abstract. For those engineers, it may be sufficient to *design* the tests without actually writing them until there is code to test. Many of the XP principles are rigidly defined, but if you understand the theory behind it, you may be able to find ways to adapt the guidelines to the needs of your project.

The collaborative aspects of XP can be challenging as well. Pair programming has measurable benefits, but it may be difficult for a manager to rationalize having half as many people actually writing code each day. Some members of the team may even feel uncomfortable with such close

collaboration, perhaps finding it difficult to type while others are watching. Pair programming also has obvious challenges if the team is physically spread out or if members tend to telecommute regularly.

For some organizations, Extreme Programming may be too radical. Large established companies with formal policies in place for engineering may be slow to adopt approaches like XP. However, even if your company is resistant to the implementation of XP, you can still improve your own productivity by understanding the theory behind it.

Software Triage

In the fatalistically-named book *Death March* (Prentice Hall, 1997) Edward Yourdon describes the frequent and scary condition of software that is behind schedule, short on staff, over budget, or poorly designed. Yourdon's theory is that when software projects get into this state, even the best modern software development methodologies will no longer apply. As you have learned in this chapter, many approaches to software development are built around formalized documents or taking a user-centered approach to design. In a project that's already in "death march" mode, there simply isn't time for these approaches.

The idea behind software triage is that when a project is already in a bad state, resources are scarce. Time is scarce, engineers are scarce, and money may be scarce. The main mental obstacle that managers and developers need to overcome when a project is way behind schedule is that it will be impossible to satisfy the original requirements in the allotted time. The task then becomes organizing remaining functionality into "must-have," "should-have," and "nice-to-have" lists.

Software triage is a daunting and delicate process. It often requires the leadership of a seasoned veteran of "death march" projects to make the tough decisions. For the engineer, the most important point is that in certain conditions, it may be necessary to throw familiar processes out the window (along with some existing code, unfortunately) to finish a project on time.

BUILDING YOUR OWN PROCESS AND METHODOLOGY

It's unlikely that any book or engineering theory will perfectly match the needs of your project or organization. I recommend that you learn from as many approaches as you can and design your own process. Combining concepts from different approaches may be easier than you think. For example, RUP optionally supports an XP-like approach. Here are some tips for building the software engineering process of your dreams.

Be Open to New Ideas

Some engineering techniques seem crazy at first or unlikely to work. Look at new innovations in software engineering methodologies as a way to refine your existing process. Try things out when you can. If XP sounds intriguing, but you're not sure if it will work in your organization, see if you can work it in slowly, taking a few of the principles at a time or trying it out with a smaller pilot project.

Bring New Ideas to the Table

Most likely, your engineering team is made up of people from varying backgrounds. You may have people who are veterans of startups, long-time consultants, recent graduates, and PhDs on your team. You all have a different set of experiences and your own ideas of how a software project should be run. Sometimes the best processes turn out to be a combination of the way things are typically done in these very different environments.

Recognize What Works and What Doesn't Work

At the end of a project (or better yet, during the project, like with the Sprint Retrospective of the Scrum methodology), get the team together to evaluate the process. Sometimes there's a major problem that nobody notices until the whole team stops to think about it. Perhaps there's a problem that *everybody* knows about but nobody has discussed.

Consider what isn't working and see how those parts can be fixed. Some organizations require formal code reviews prior to any source code check-in. If code reviews are so long and boring that nobody does a good job, discuss code-reviewing techniques as a group. For example, consider reviewing only the interface files. Your group might even decide that changing an interface file after the interface is finalized with a review will be marked as a failure. This might help your team members to quickly learn how to write good interfaces.

Also consider what is going well and see how those parts can be extended. For example, if maintaining the feature tasks as a group-editable website is working, then maybe devote some time to making the website better.

Don't Be a Renegade

Whether a process is mandated by your manager or custom-built by the team, it's there for a reason. If your process involves writing formal design documents, make sure you write them. If you think that the process is broken or too complex, see if you can talk to your manager about it. Don't just avoid the process — it will come back to haunt you.

SOURCE CODE CONTROL

Managing all source code is very important for any company, big or small, even for personal developers. In a company for example, it would be very impractical to store all the source code on the machines of the individual developers not managed by any *Source Code Control* software. This would result in a maintenance nightmare because not everyone will always have the latest code. All source code must be managed by Source Code Control software. There are three kinds of Source Code Control software solutions:

- **Local:** These solutions store all source code files and their history locally on your machine and are not really suitable for use in a team. These are solutions from the 70s and 80s and shouldn't be used anymore. They are not discussed further.
- **Client/Server:** These solutions are split into a client component and a server component. For a personal developer, the client and server components can run on the same machine,

but the separation makes it easy to move the server component to a dedicated physical server machine.

- **Distributed:** These solutions go one step further than the client/server model. There is no central place where everything is stored. Every developer has a copy of all the files including all the history. A peer-to-peer approach is used instead of a client/server approach. Code is synchronized between peers by exchanging patches.

The client/server solution consist of two parts. The first part is the server software, which is software running on the central server and which is responsible for keeping track of all source code files and their history. The second part is the client software. This client software should be installed on every developer's machine and is responsible to communicate with the server software to get the latest version of a source file, get a previous version of a source file, commit local changes back to the server, rollback changes to a previous version and so on.

A distributed solution doesn't use a central server. The client software uses peer-to-peer protocols to synchronize with other peers by exchanging patches. Common operations such as committing changes, rolling back changes and so on, are fast because no network access to a central server is involved. The disadvantage is that it requires more space on the client machine because it needs to store all the files, including the entire history.

Most Source Code Control systems have a special terminology, but unfortunately, not all systems use exactly the same terms. The following list explains a number of terms that are commonly used:

- **Branch:** The source code can be *branched*, which means that multiple versions can be developed side-by-side. For example, one branch could be created for every released version. On those branches, bug fixes could be implemented for those released versions, while new features are added to the main branch. Bug fixes created for released versions can also be merged back to the main branch.
- **Checkout:** This is the action of creating a local copy on the developer's machine, either coming from a central server or from peers.
- **Checkin, Commit, or Merge:** A developer should make changes to the local copy of the source code. When everything works correctly on the local machine, the developer can checkin/commit/merge those local changes back to the central server, or exchange patches with peers.
- **Conflict:** When multiple developers make changes to the same source file, a conflict might occur during checkin of that source file. The Source Code Control software often will try to automatically resolve these conflicts. If that is not possible, the client software will ask the user to resolve any conflicts manually.
- **Label or Tag:** A label or tag can be attached to all files or to a specific commit at any given time. This makes it easy to jump back to the version of the source code at that time.
- **Repository:** The collection of all files managed by the Source Code Control software is called the repository. This also includes metadata about those files, such as checkin comments.

- **Resolve:** When checkin conflicts occur, the user will have to resolve them before checkin can continue.
- **Revision or Version:** A revision or version is a snapshot of the contents of a file at a specific point in time. Versions represent specific points to which the code can be reverted to, or compared against.
- **Update or Sync:** Updating or synchronizing means that the local copy on the developer's machine is synchronized with a version on the central server or with peers. Note that this may require a merge, which may result in a conflict that needs to be resolved.
- **Working Copy:** The working copy is the local copy on the individual developer's machine.

Several Source Code Control software solutions are available. Some of them are free, some of them are commercial. The following table lists a few available solutions:

	FREE/OPEN-SOURCE	COMMERCIAL
LOCAL ONLY	SCCS, RCS	PVCS
CLIENT/SERVER	CVS, Subversion	IBM Rational ClearCase, Microsoft Team Foundation Server, Perforce
DISTRIBUTED	Git, Mercurial, Bazaar	TeamWare, BitKeeper, Plastic SCM

NOTE *The preceding list is definitely not an exhaustive list. It's just a small selection to give you an idea of what's available.*

This book does not recommend one or the other software solution. Most software companies these days have a Source Code Control solution already in place, which every developer needs to adopt. If this is not the case, the company should definitely invest some time for doing research on the available solutions, and to pick one that suits them. The bottom line is that it will be a maintenance nightmare without any Source Code Control solution in place. Even for your personal projects you might want to investigate the available solutions. If you find one that you like to work with, it will make your life easier. It will automatically keep track of different versions and a history of your changes. This makes it easy for you to change back to an older version if a change didn't work out the way it was supposed to.

SUMMARY

This chapter introduced you to several models and methodologies for the software development process. There are certainly many other ways of building software, both formalized and informal. There probably isn't a single correct method for developing software except the method that

works for your team. The best way to find this method is to do your own research, learn what you can from various methods, talk to your peers about their experiences, and iterate on your process. Remember, the only metric that matters when examining a process methodology is how much it helps your team to write code.

The last part of this chapter briefly touched on the concept of Source Code Control. This should be an integral part of any software company, big or small, and can even be beneficial for personal projects at home. There are several Source Code Control software solutions available, so it is recommended that you try out a few and see which of them works for you.

25

Writing Efficient C++

WHAT'S IN THIS CHAPTER?

- What “efficiency” and “performance” mean
- What kind of language-level optimizations you can use
- Which design-level guidelines you can follow to design efficient programs
- What profiling tools are

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter’s code download on the book’s website at www.wrox.com/go/proc++3e on the Download Code tab.

The efficiency of your programs is important regardless of your application domain. If your product competes with others in the marketplace, speed can be a major differentiator: Given the choice between a slower and a faster program, which one would you choose? No one would buy an operating system that takes two weeks to boot up, unless it was the only option. Even if you don’t intend to sell your products, they will have users. Those users will not be happy with you if they end up wasting time waiting for your programs to complete tasks.

Now that you understand the concepts of Professional C++ design and coding, and have tackled some of the more complex facilities that the language provides, you are ready to incorporate performance into your programs. Writing efficient programs involves thought at the design level, as well as details at the implementation level. Although this chapter falls late in this book, remember to consider performance from the beginning of your program life cycle.

OVERVIEW OF PERFORMANCE AND EFFICIENCY

Before delving further into the details, it's helpful to define the terms performance and efficiency, as used in this book. The *performance* of a program can refer to several areas, such as speed, memory usage, disk access, and network use. This chapter focuses on speed performance. The term *efficiency*, when applied to programs, means running without wasted effort. An efficient program completes its tasks as quickly as possible within the given circumstances. A program can be efficient without being fast, if the application domain is inherently prohibitive to quick execution.

NOTE *An efficient, or high-performance, program runs as fast as is possible for the particular tasks.*

Note that the title of this chapter, “Writing Efficient C++,” means writing programs that run efficiently, not efficiently writing programs. That is, the time you learn to save by reading this chapter will be your users’, not your own!

Two Approaches to Efficiency

Language-level efficiency involves using the language as efficiently as possible, for example, passing objects by reference instead of by value. However, this will only get you so far. Much more important is *design-level efficiency*, which includes choosing efficient algorithms, avoiding unnecessary steps and computations, and selecting appropriate design optimizations. Optimizing existing code, more often than not, involves replacing a bad algorithm or data structure with a better, more efficient one.

Two Kinds of Programs

As noted, efficiency is important for all application domains. Additionally, there is a small subset of programs, such as system-level software, embedded systems, intensive computational applications, and real-time games, which require extremely high levels of efficiency. Most programs don't. Unless you write those types of high-performance applications, you probably don't need to worry about squeezing every ounce of speed out of your C++ code. Think of it as the difference between building normal family cars and building sports cars. Every car must be reasonably efficient, but sports cars require extremely high performance. You wouldn't want to waste your time optimizing family cars for speed when they'll never go faster than 70 miles per hour.

Is C++ an Inefficient Language?

C programmers often resist using C++ for high-performance applications. They claim that the language is inherently less efficient than C or a similar procedural language because C++ includes high-level concepts, such as exceptions and virtual methods. However, there are problems with this argument.

First, you cannot ignore the effect of compilers. When discussing the efficiency of a language, you must separate the performance capabilities of the language itself from the effectiveness of its compilers at optimizing it. Recall that the C or C++ code you write is not the code that the computer

executes. A compiler first translates that code into machine language, applying optimizations in the process. This means that you can't simply run benchmarks of C and C++ programs and compare the result. You're really comparing the compiler optimizations of the languages, not the languages themselves. C++ compilers can "optimize away" many of the high-level constructs in the language to generate machine code similar to that generated from a comparable C program.

Critics, however, still maintain that some features of C++ cannot be optimized away. For example, as Chapter 9 explains, virtual methods require the existence of a vtable and an additional level of indirection at run time, possibly making them slower than regular non-virtual function calls. However, when you really think about it, this argument is still unconvincing. Virtual method calls provide more than just a function call: They also give you a run-time choice of which function to call. A comparable non-virtual function call would need a conditional statement to decide which function to call. If you don't need those extra semantics, you can use a non-virtual function. A general design rule in the C++ language is, "if you don't use it, you don't need to pay for it." If you don't use virtual methods, you pay no performance penalty for the fact that you could use them. Thus, non-virtual function calls in C++ are identical to function calls in C in terms of performance. However, since virtual function calls have such a tiny overhead, I recommend making all your class methods, including destructors but not constructors, virtual for all your non-final classes.

Far more important, the high-level constructs of C++ enable you to write cleaner programs that are more efficient at the design level, are more easily maintained, and avoid accumulating unnecessary and dead code.

I believe that you will be better served in your development, performance, and maintenance by choosing C++ instead of a procedural language.

LANGUAGE-LEVEL EFFICIENCY

Many books, articles, and programmers spend a lot of time trying to convince you to apply language-level optimizations to your code. These tips and tricks are important, and can speed up your programs in some cases. However, they are far less important than the overall design and algorithm choices in your program. You can pass-by-reference all you want, but it won't ever make your program fast if you perform twice as many disk writes as you need. It's easy to get bogged down in references and pointers and forget about the big picture.

Furthermore, some of these language-level tricks can be performed automatically by good optimizing compilers. You should never spend time optimizing a particular area, unless a profiler, discussed later in this chapter, tells you that that particular area is a bottleneck.

That being said, using certain language-level optimizations, such as pass-by-reference, is just considered good coding style.

In this book, I've tried to present a balance of strategies. Thus, I've included here what I feel are the most useful language-level optimizations. This list is not comprehensive, but should give you a good start if you want to optimize your code. However, make sure to read, and practice, the design-level efficiency advice described later in this chapter as well.

WARNING *Apply language-level optimizations judiciously. I recommend to make a clean, well-structured design and implementation first. Then use a profiler, and only invest time optimizing those parts that are flagged by a profiler as being a performance bottleneck.*

Handle Objects Efficiently

C++ does a lot of work for you behind the scenes, particularly with regard to objects. You should always be aware of the performance impact of the code you write. If you follow a few simple guidelines, your code will become more efficient. Note that these guidelines are only relevant for objects, and not for primitive types such as `bool`, `int`, `float`, and so on.

Pass-by-Reference

This rule is discussed elsewhere in this book, but it's worth repeating here.

WARNING *Objects should rarely be passed by value to a function or method.*

Pass-by-value incurs copying costs that are avoided with pass-by-reference. One reason why this rule can be difficult to remember is that on the surface there doesn't appear to be any problem when you pass-by-value. Consider a class to represent a person that looks like this:

```
class Person
{
public:
    Person();
    Person(const std::string& inFirstName, const std::string& inLastName,
           int inAge);
    const std::string& getFirstName() const { return mFirstName; }
    const std::string& getLastName() const { return mLastName; }
    int getAge() const { return mAge; }
private:
    std::string mFirstName, mLastName;
    int mAge;
};
```

You could write a function that takes a `Person` object as follows:

```
void processPerson(Person p)
{
    // Process the person.
}
```

You might call it like this:

```
Person me("Marc", "Gregoire", 35);
processPerson(me);
```

This doesn't look like there's any more code than if you wrote the function like this instead:

```
void processPerson(const Person& p)
{
    // Process the person.
}
```

The call to the function remains the same. However, consider what happens when you pass-by-value in the first version of the function. In order to initialize the `p` parameter of `processPerson()`, `me` must be copied with a call to its copy constructor. Even though you didn't write a copy constructor for the `Person` class, the compiler generates one that copies each of the data members. That still doesn't look so bad: there are only three data members. However, two of those are strings, which are themselves objects with copy constructors. So, each of their copy constructors will be called as well. The version of `processPerson()` that takes `p` by reference incurs no such copying costs. Thus, pass-by-reference in this example avoids three constructor calls when the code enters the function.

And you're still not done. Remember that `p` in the first version of `processPerson()` is a local variable to the `processPerson()` function, and so must be destroyed when the function exits. This destruction requires a call to the `Person` destructor, which will call the destructor of all of the data members. `strings` have destructors, so exiting this function (if you passed by value) incurs calls to three destructors. None of those calls are needed if the `Person` object is passed by reference.

NOTE *If a function must modify an object, you can pass the object by reference. If the function should not modify the object, you can pass it by const reference, as in the preceding example. See Chapter 10 for details on references and const.*

NOTE *Avoid using pass-by-pointer, which is a relatively obsolete method for pass-by-reference, and is a throwback to the C language, rarely suitable for C++ (unless passing `nullptr` has meaning in your design).*

Return-by-Reference

Just as you should pass objects by reference to functions, you should also return them by reference from functions in order to avoid copying the objects unnecessarily. Unfortunately, it is sometimes impossible to return objects by reference, such as when you write overloaded `operator+` and other similar operators. You should never return a reference or a pointer to a local object that will be destroyed when the function exits.

Since C++11, the language has support for move semantics, which allows you to efficiently return objects by value, instead of using reference semantics.

Catch Exceptions by Reference

As noted in Chapter 13, you should catch exceptions by reference in order to avoid an extra copy. Throwing exceptions is heavy in terms of performance, so any little thing you can do to improve their efficiency will help.

Use Move Semantics

You should implement a move constructor and move assignment operator for your objects, which allow the C++ compiler to use move semantics. With move semantics for your objects, returning them by value from a function will be efficient without incurring large copying costs. Consult Chapter 10 for details on move semantics.

Avoid Creating Temporary Objects

The compiler creates temporary, unnamed objects in several circumstances. Chapter 8 explains that after writing a global `operator+` for a class, you can add objects of that class to other types, as long as those types can be converted to objects of that class. For example, the `SpreadsheetCell` class definition looks in part like the following:

```
class SpreadsheetCell
{
public:
    // Other constructors omitted for brevity
    SpreadsheetCell(double initialValue);
    friend SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    // Remainder omitted for brevity
};
```

The constructor that takes a `double` allows you to write code like this:

```
SpreadsheetCell myCell(4), aThirdCell;
aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

The first addition line constructs a temporary `SpreadsheetCell` object from the `5.6` argument; then calls the `operator+` with `myCell` and the temporary object as arguments. The result is stored in `aThirdCell`. The second addition line does the same thing, except that `4` must be coerced to a `double` in order to call the `double` constructor of the `SpreadsheetCell`.

The important point in this example is that the compiler generates code to create an extra, unnamed `SpreadsheetCell` object for each addition line. That object must be constructed and destructed with calls to its constructor and destructor. If you're still skeptical, try inserting `cout` statements in your constructor and destructor and watching the printout.

In general, the compiler constructs a temporary object whenever your code converts a variable of one type to another type for use in a larger expression. This rule applies mostly to function calls. For example, suppose that you write a function with the following prototype:

```
void doSomething(const SpreadsheetCell& s);
```

You can call it like this:

```
doSomething(5.56);
```

The compiler constructs a temporary `SpreadsheetCell` object from `5.56` using the `double` constructor, which it passes to `doSomething()`. Note that if you remove the `const` from the `s` parameter, you can no longer call `doSomething()` with a constant; you must pass a variable.

You should generally attempt to avoid cases in which the compiler is forced to construct temporary objects. Although it is impossible to avoid in some situations, you should at least be aware of the existence of this “feature” so you aren’t surprised by performance and profiling results.

Move semantics are also used by the compiler to make working with temporary objects more efficient. That’s another reason to add move semantics to your classes.

The Return-Value Optimization

A function that returns an object by value can cause the creation of a temporary object. Continuing with the `Person` example, consider this function:

```
Person createPerson()
{
    Person newP;
    return newP;
}
```

Suppose that you call it like this (assuming that `operator<<` is implemented for the `Person` class):

```
cout << createPerson();
```

Even though this call does not store the result of `createPerson()` anywhere, the result must be stored somewhere in order to pass it to `operator<<`. In order to generate code for this behavior, the compiler is allowed to create a temporary variable in which to store the `Person` object returned from `createPerson()`.

Even if the result of the function is not used anywhere, the compiler might still generate code to create the temporary object. For example, suppose that you have this code:

```
createPerson();
```

The compiler might generate code to create a temporary object for the return value, even though it is not used.

However, you usually don’t need to worry about this issue because the compiler optimizes away the temporary variable in most cases. This optimization is called the *return-value optimization* and is usually only enabled for release builds.

Note that if the object you want to return from a function supports move semantics, then it is moved out of the function instead of copied.

Use Inline Methods and Functions

As described in Chapter 8, the code for an `inline` method or function is inserted directly into the code where it is called, avoiding the overhead of a function call. You should mark as `inline` all functions and methods that you think can qualify for this optimization. However, remember that inlining requests by the programmer are only a recommendation to the compiler, which is allowed to refuse them.

On the other hand, some compilers inline appropriate functions and methods during their optimization steps, even if those functions aren't marked with the `inline` keyword. Thus, you should read your compiler documentation before wasting a lot of effort deciding which functions to inline.

DESIGN-LEVEL EFFICIENCY

The design choices in your program affect its performance far more than do language details such as pass-by-reference. For example, if you choose an algorithm for a fundamental task in your application that runs in $O(n^2)$ time instead of a simpler one that runs in $O(n)$ time, you could potentially perform the square of the number of operations that you really need. To put numbers on that, a task that uses an $O(n^2)$ algorithm and performs one million operations would perform only one thousand with an $O(n)$ algorithm. Even if that operation is optimized beyond recognition at the language level, the simple fact that you perform one million operations when a better algorithm would use only one thousand will make your program very inefficient. Always choose your algorithms carefully. Refer to Part II, specifically Chapter 4, of this book for a detailed discussion of algorithm design choices and big-O notation.

In addition to your choice of algorithms, design-level efficiency includes specific tips and tricks. Instead of writing your own data structures and algorithms, you should use existing ones, such as the STL or the Boost libraries, as much as possible because these are written by experts. You should also think about incorporating multithreading in your design, see Chapter 23. The remainder of this section presents two more design techniques for optimizing your program: caching, and object pools.

Cache Where Necessary

Caching means storing items for future use in order to avoid retrieving or recalculating them. You might be familiar with the principle from its use in computer hardware. Modern computer processors are built with memory caches that store recently and frequently accessed memory values in a location that is quicker to access than main memory. Most memory locations that are accessed at all are accessed more than once in a short time period, so caching at the hardware level can significantly speed up computations.

Caching in software follows the same approach. If a task or computation is particularly slow, you should make sure that you are not performing it more than necessary. Store the results in memory the first time you perform the task so that they are available for future needs. Here is a list of tasks that are usually slow:

- **Disk access:** You should avoid opening and reading the same file more than once in your program. If memory is available, save the file contents in RAM if you need to access it frequently.

- **Network communication:** Whenever you need to communicate over a network, your program is subject to the vagaries of the network load. Treat network accesses like file accesses, and cache as much static information as possible.
- **Mathematical computations:** If you need the result of a very complex computation in more than one place, perform the calculation once and share the result. However, if it's not very complex, then it's probably faster to just calculate it instead of retrieving it from a cache. Use a profiler if you want to make sure.
- **Object allocation:** If you need to create and use a large number of short-lived objects in your program, consider using an object pool, described later in this chapter.
- **Thread creation:** This task can also be slow. You can “cache” threads in a thread-pool, similar to caching objects in an object-pool.

Cache Invalidation

One common problem with caching is that the data you store are often only copies of the underlying information. The original data might change during the lifetime of the cache. For example, you might want to cache the values in a configuration file so that you don't need to read it repeatedly. However, the user might be allowed to change the configuration file while your program is running, which would make your cached version of the information obsolete. In cases like this, you need a mechanism for *cache invalidation*: When the underlying data change, you must either stop using your cached information, or repopulate your cache.

One technique for cache invalidation is to request that the entity managing the underlying data notifies your program of the data change. It could do this through a *callback* that your program registers with the manager. Alternatively, your program could poll for certain events that would trigger it to repopulate the cache automatically. Regardless of your specific cache invalidation technique, make sure you think about these issues before relying on a cache in your program.

NOTE Always keep in mind that maintaining caches takes code, memory, and processing time. On top of that, caches can be a source of subtle bugs. You should only add caching to a particular area when a profiler clearly shows that that area is a performance bottleneck. First write clean and correct code, then profile it, and only then optimize parts of it.

Use Object Pools

If your program needs a large number of short-lived objects of the same type that have an expensive constructor (for example, a constructor creating a large, pre-sized vector for storing data), and a profiler confirms that allocating and deallocating these objects is a bottleneck, then you can create a *pool*, or cache, of those objects. Whenever you need an object in your code, you ask the pool for one. When you are done with the object, you return it to the pool. The object pool creates the objects only once, so their constructor is called only once, not each time they are used. Thus, object pools are appropriate when the constructor performs some setup actions that apply to many uses of the object, and when you can set instance-specific parameters on the object through non-constructor method calls.

An Object Pool Implementation

This section provides an implementation of a pool class template that you can use in your programs. The pool allocates a *chunk* of objects of the specified class when it is constructed and hands them out via the `acquireObject()` method. If `acquireObject()` is called but there are no free objects, the pool allocates another chunk of objects. `acquireObject()` returns an `Object` which is a `std::shared_ptr` with a custom deleter. The custom deleter doesn't actually delete the memory but it simply puts the object back on the list of free objects.

The most difficult aspect of an object pool implementation is keeping track of which objects are free and which are in use. This implementation takes the approach of storing free objects in a queue. Each time a client requests an object, the pool gives that client the top object from the queue.

The code uses the `std::queue` class from the Standard Template Library (STL), discussed in Chapter 16. Memory for allocated objects is handled by `std::unique_ptr` and `std::shared_ptr` smart pointers.

Here is the class definition, with comments that explain the details. Note that the template is parameterized on the class type from which the objects in the pool are to be constructed.

```
#include <cstddef>
#include <queue>
#include <stdexcept>
#include <memory>

// Provides an object pool that can be used with any class that provides a
// default constructor.
//
// The object pool constructor creates a pool of objects, which it hands out
// to clients when requested via the acquireObject() method. acquireObject()
// returns an Object which is a std::shared_ptr with a custom deleter that
// automatically puts the object back into the object pool when the shared_ptr
// is destroyed and its reference reaches 0.
//
// The constructor and destructor on each object in the pool will be called only
// once each for the lifetime of the program, not once per acquisition and release.
//
// The primary use of an object pool is to avoid creating and deleting objects
// repeatedly. The object pool is most suited to applications that use large
// numbers of objects with expensive constructors for short periods of time, if
// a profiler tells you that allocating and deallocating these objects is a
// bottleneck.
template <typename T>
class ObjectPool
{
public:
    // Creates an object pool with chunkSize objects.
    // Whenever the object pool runs out of objects, chunkSize
    // more objects will be added to the pool. The pool only grows:
    // objects are never removed from the pool (freed), until
    // the pool is destroyed.
    //
    // Throws invalid_argument if chunkSize is 0.
```

```

    // Throws bad_alloc if allocation fails.
    ObjectPool(size_t chunkSize = kDefaultChunkSize);

    // Prevent assignment and pass-by-value
    ObjectPool(const ObjectPool<T>& src) = delete;
    ObjectPool<T>& operator=(const ObjectPool<T>& rhs) = delete;

    // The type of smart pointer returned by acquireObject().
    using Object = std::shared_ptr<T>;

    // Reserves an object for use.
    Object acquireObject();

private:
    // mFreeList stores the objects that are not currently in use by clients.
    std::queue<std::unique_ptr<T>> mFreeList;
    size_t mChunkSize;
    static const size_t kDefaultChunkSize = 10;
    // Allocates mChunkSize new objects and adds them to mFreeList.
    void allocateChunk();
};

}

```

Note that the user of the object pool specifies through the template parameter the name of the class from which objects can be created, and through the constructor the allocation “chunk size.” This “chunk size” controls the number of objects created at one time. Here is the code that defines the `kDefaultChunkSize`:

```

template<typename T>
const size_t ObjectPool<T>::kDefaultChunkSize;

```

The default of 10, given in the class definition, is probably too small for most uses. If your program requires thousands of objects at once, you should use a larger, more appropriate, value.

The constructor validates the `chunkSize` parameter, and calls the `allocateChunk()` helper method to obtain a starting allocation of objects:

```

template <typename T>
ObjectPool<T>::ObjectPool(size_t chunkSize)
{
    if (chunkSize == 0) {
        throw std::invalid_argument("chunk size must be positive");
    }
    mChunkSize = chunkSize;
    // Create mChunkSize objects to start.
    allocateChunk();
}

```

The `allocateChunk()` method allocates `mChunkSize` elements. It stores a `unique_ptr` to each object in the queue:

```

// Allocates mChunkSize new objects.
template <typename T>

```

```
void ObjectPool<T>::allocateChunk()
{
    for (size_t i = 0; i < mChunkSize; ++i) {
        mFreeList.emplace(std::make_unique<T>());
    }
}
```

`acquireObject()` returns the top object from the free list, first calling `allocateChunk()` if there are no free objects:

```
template <typename T>
typename ObjectPool<T>::Object ObjectPool<T>::acquireObject()
{
    if (mFreeList.empty()) {
        allocateChunk();
    }

    // Move next free object from the queue to a local unique_ptr.
    std::unique_ptr<T> obj(std::move(mFreeList.front()));
    mFreeList.pop();

    // Convert the object pointer to an Object (a shared_ptr with
    // a custom deleter).
    Object smartObject(obj.release(), [this](T* t) {
        // The custom deleter doesn't actually deallocate the
        // memory, but simply puts the object back on the free list.
        mFreeList.push(std::unique_ptr<T>(t));
    });

    // Return the Object.
    return smartObject;
}
```

Using the Object Pool

Consider an application that is structured around obtaining requests for actions from users and processing those requests. This application would most likely be the middleware between a graphical front-end and a back-end database. For example, it could be part of an airline reservation system or an online banking application. You might want to encode each user request in an object, with a class that looks something like this:

```
class UserRequest
{
public:
    UserRequest() {}
    virtual ~UserRequest() {}
    // Methods to populate the request with specific information.
    // Methods to retrieve the request data.
    // (not shown)
private:
    // Data members (not shown)
};
```

Instead of creating and deleting large numbers of requests throughout the lifetime of your program, you could use an object pool. Your program structure could be something as follows:

```
ObjectPool<UserRequest>::Object obtainUserRequest (ObjectPool<UserRequest>& pool)
{
    // Obtain a UserRequest object from the pool.
    auto request = pool.acquireObject();
    // Populate the request with user input. (not shown)
    return request;
}

void processUserRequest (ObjectPool<UserRequest>::Object& req)
{
    // Process the request. (not shown)
    // Return the request to the pool.
    req.reset();
}

int main()
{
    ObjectPool<UserRequest> requestPool(10);
    for (size_t i = 0; i < 100; ++i) {
        auto req = obtainUserRequest(requestPool);
        processUserRequest(req);
    }
    return 0;
}
```

PROFILING

It is good to think about efficiency as you design and code. There is no point in writing obviously inefficient programs if it can be avoided with some common sense, or experience-based intuitions. However, I urge you not to get too obsessed with performance during the design and coding phases. It's best to first make a clean, well-structured design and implementation, then use a profiler, and only optimize parts that are flagged by the profiler as being performance bottlenecks. Remember, Chapter 4 introduces the “90/10” rule: 90 percent of the running time of most programs is spent in only 10 percent of the code (Hennessy and Patterson, *Computer Architecture, A Quantitative Approach, Fourth Edition*, [Morgan Kaufmann, 2006]). This means that you could optimize 90 percent of your code out of existence, but still only improve the running time of the program by 10 percent. Obviously, you want to optimize the parts of the code that are exercised the most for the specific workload that you expect the program to run.

Consequently, it is often helpful to *profile* your program to determine which parts of the code require optimization. There are many *profiling tools* available that analyze programs as they run in order to generate data about their performance. Most profiling tools provide analysis at the function level by specifying the amount of time (or percent of total execution time) spent in each function in the program. After running a profiler on your program, you can usually tell immediately which parts of the program need optimization. Profiling before and after optimizing is also useful to prove that your optimizations had an effect.

If you are using Microsoft Visual C++ 2013, you already have a great built-in profiler, which is discussed later in this chapter. Another great profiling tool is Rational PurifyPlus from IBM. Both of these products require license fees, but you should check if your company or academic institution has a site license for their use. If the license restriction is prohibitive, there are several free profiling tools. Very Sleepy and Luke Stackwalker are popular profilers for Windows, Valgrind and gprof (GNU profiler) are well-known profilers for Unix/Linux systems, and there are plenty of other choices.

Profiling Example with gprof

The power of profiling can best be seen with a real coding example. As a disclaimer, the performance bugs in the first attempt shown are not subtle. Real efficiency issues would probably be more complex, but a program long enough to demonstrate them would be too lengthy for this book.

Suppose that you work for the United States Social Security Administration. Every year the administration puts up a website that allows users to look up the popularity of new baby names from the previous year. Your job is to write the back-end program that looks up names for users. Your input is a file containing the name of every new baby. This file will obviously contain redundant names. For example, in the file for boys for 2003, the name Jacob was the most popular, showing up 29,195 times. Your program must read the file to construct an in-memory database. A user may then request the absolute number of babies with a given name, or the rank of that name among all the babies.

First Design Attempt

A logical design for this program consists of a `NameDB` class with the following public methods:

```
#include <string>

class NameDB
{
public:
    // Reads the list of baby names in nameFile to populate the database.
    // Throws invalid_argument if nameFile cannot be opened or read.
    NameDB(const std::string& nameFile);
    // Returns the rank of the name (1st, 2nd, etc).
    // Returns -1 if the name is not found.
    int getNameRank(const std::string& name) const;
    // Returns the number of babies with this name.
    // Returns -1 if the name is not found.
    int getAbsoluteNumber(const std::string& name) const;
    // Protected and private members and methods not shown
};
```

The hard part is choosing a good data structure for the in-memory database. A first attempt might be an array, or a `vector` from the STL, of name/count pairs. Each entry in the `vector` would store one of the names, along with a count of the number of times that name shows up in the raw data file. Here is the complete class definition with such a design:

```

#include <string>
#include <vector>
#include <utility>

class NameDB
{
public:
    NameDB(const std::string& nameFile);
    int getNameRank(const std::string& name) const;
    int getAbsoluteNumber(const std::string& name) const;
private:
    std::vector<std::pair<std::string, int>> mNames;
    // Helper methods
    bool nameExists(const std::string& name) const;
    void incrementNameCount(const std::string& name);
    void addNewName(const std::string& name);
};

```

Note the use of the STL vector and pair, discussed in Chapter 16. A pair is a utility class that combines two variables of different types.

Here are the implementations of the constructor and the helper methods `nameExists()`, `incrementNameCount()`, and `addNewName()`. The loops in `nameExists()` and `incrementNameCount()` iterate over all the elements of the vector.

```

// Reads the names from the file and populates the database.
// The database is a vector of name/count pairs, storing the
// number of times each name shows up in the raw data.
NameDB::NameDB(const string& nameFile)
{
    // Open the file and check for errors.
    ifstream inFile(nameFile.c_str());
    if (!inFile) {
        throw invalid_argument("Unable to open file");
    }
    // Read the names one at a time.
    string name;
    while (inFile >> name) {
        // Look up the name in the database so far.
        if (nameExists(name)) {
            // If the name exists in the database, just increment the count.
            incrementNameCount(name);
        } else {
            // If the name doesn't yet exist, add it with a count of 1.
            addNewName(name);
        }
    }
    inFile.close();
}

// Returns true if the name exists in the database, false otherwise.
bool NameDB::nameExists(const string& name) const
{
    // Iterate through the vector of names looking for the name.

```

```

        for (auto& entry : mNames) {
            if (entry.first == name) {
                return true;
            }
        }
        return false;
    }

    // Precondition: name exists in the vector of names.
    // Postcondition: the count associated with name is incremented.
    void NameDB::incrementNameCount(const string& name)
    {
        for (auto& entry : mNames) {
            if (entry.first == name) {
                entry.second++;
                return;
            }
        }
    }

    // Adds a new name to the database.
    void NameDB::addNewName(const string& name)
    {
        mNames.push_back(make_pair(name, 1));
    }
}

```

Note that in the preceding example, you could use an algorithm like `find_if()` to accomplish the same thing as the loops in `nameExists()` and `incrementNameCount()`. The loops are shown explicitly in order to emphasize the performance problems.

The savvy reader might notice some performance problems already. What if there are hundreds of thousands of names? The many linear searches involved in populating the database might become slow.

In order to complete the example, here are the implementations of the two public methods:

```

    // Returns the rank of the name.
    // First looks up the name to obtain the number of babies with that name.
    // Then iterates through all the names, counting all the names with a higher
    // count than the specified name. Returns that count as the rank.
    int NameDB::getNameRank(const string& name) const
    {
        // Make use of the getAbsoluteNumber() method.
        int num = getAbsoluteNumber(name);
        // Check if we found the name.
        if (num == -1) {
            return -1;
        }
        // Now count all the names in the vector that have a
        // count higher than this one. If no name has a higher count,
        // this name is rank number 1. Every name with a higher count
        // decreases the rank of this name by 1.
        int rank = 1;
        for (auto& entry : mNames) {
            if (entry.second > num) {

```

```

        rank++;
    }
}
return rank;
}

// Returns the count associated with this name.
int NameDB::getAbsoluteNumber(const string& name) const
{
    for (auto& entry : mNames) {
        if (entry.first == name) {
            return entry.second;
        }
    }
    return -1;
}

```

Profile of the First Attempt

In order to test the program, you need a `main()` function:

```

#include "NameDB.h"
#include <iostream>
using namespace std;
int main()
{
    NameDB boys("boys_long.txt");
    cout << boys.getNameRank("Daniel") << endl;
    cout << boys.getNameRank("Jacob") << endl;
    cout << boys.getNameRank("William") << endl;
    return 0;
}

```

This `main()` function creates one `NameDB` database called `boys`, telling it to populate itself with the file `boys_long.txt`, which contains 500,500 names.

There are three steps to using `gprof`:

1. Compile your program with a special flag that causes it to log raw execution information next time it is run. When using GCC as your compiler, the flag is `-pg`, for example:


```
> gcc -lstdc++ -std=c++11 -pg -o namedb NameDB.cpp NameDBTest.cpp
```
2. Next, run your program. This run should generate a file called `gmon.out` in the working directory. Be patient when you run the program because this first version is slow.


```
> ./namedb
```
3. The final step is to run the `gprof` command in order to analyze the `gmon.out` profiling information and produce a (somewhat) readable report. `gprof` outputs to standard out, so you should redirect the output to a file:


```
> gprof namedb gmon.out > gprof_analysis.out
```

Now you can analyze the data. Unfortunately, the output file is somewhat cryptic and intimidating. It takes a little while to learn how to interpret it. `gprof` provides two separate sets of information.

The first set summarizes the amount of time spent executing each function in the program. The second, and more useful set summarizes the amount of time spent executing each function *and its descendants*, and is also called a *call graph*. Here is some of the output from the `gprof_analysis.out` file, edited to make it more readable. Note that the numbers will be different on your machine.

index	%time	self	children	called	name
[1]	100.0	0.00	14.06		main [1]
		0.00	14.00	1/1	NameDB::NameDB [2]
		0.00	0.04	3/3	NameDB::getNameRank [25]
		0.00	0.01	1/1	NameDB::~NameDB [28]

The following list explains the different columns:

- `index`: an index to be able to refer to this entry in the call graph.
- `%time`: the percentage of the total execution time of the program required by this function and its descendants.
- `self`: how many seconds the function itself was executing.
- `children`: how many seconds the descendants of this function were executing.
- `called`: how often this function was called.
- `name`: the name of the function. If the name of the function is followed by a number between square brackets, that number refers to another index in the call graph.

The preceding extract tells us that `main()` and its descendants took 100 percent of the total execution time of the program, for a total of 14.06 seconds. The second line shows that the `NameDB` constructor took 14.00 seconds of the total 14.06 seconds. So it's immediately clear where the performance issue is situated. To track down which part of the constructor is taking so long, you need to jump to the call graph entry with index 2, because that's the index in square brackets behind the name in the last column. The call graph entry with index 2 is as follows on my test system:

[2]	99.6	0.00	14.00	1	NameDB::NameDB [2]
	1.20	6.14	500500/500500		NameDB::nameExists [3]
	1.24	5.24	499500/499500		NameDB::incrementNameCount [4]
	0.00	0.18	1000/1000		NameDB::addNewName [19]
	0.00	0.00	1/1		vector::vector [69]

The nested entries below `NameDB::NameDB` show which of its descendants took the most time. Here you can see that `nameExists()` took 6.14 seconds, and `incrementNameCount()` took 5.24 seconds. Remember that these times are the sums of all the calls to the functions. The fourth column in those lines shows the number of calls to the function (500,500 to `nameExists()` and 499,500 to `incrementNameCount()`). No other function took a significant amount of time.

Without going any further in this analysis, two things should jump out at you:

1. 14 seconds to populate the database of approximately 500,000 names is slow. Perhaps you need a better data structure.
2. `nameExists()` and `incrementNameCount()` take almost identical time, and are called almost the same number of times. If you think about the application domain, that makes sense: Most names in the input text file are duplicates, so the vast majority of the calls to

`nameExists()` are followed by a call to `incrementNameCount()`. If you look back at the code, you can see that these functions are almost identical; they could probably be combined. In addition, most of what they are doing is searching the vector. It would probably be better to use a sorted data structure to reduce the searching time.

Second Attempt

With these two observations from the gprof output, it's time to redesign the program. The new design uses a `map` instead of a `vector`. Chapter 16 explains that the STL `map` keeps the entries sorted, and provides $O(\log n)$ lookup instead of the $O(n)$ searches in a `vector`. A good exercise for the reader is to use a `std::unordered_map`, which has an expected $O(1)$ for lookups, and use a profiler to see if that is faster than `std::map` for this application.

The new version of the program also combines `nameExists()` and `incrementNameCount()` into one `nameExistsAndIncrement()` method.

Here is the new class definition:

```
#include <string>
#include <map>

class NameDB
{
public:
    NameDB(const std::string& nameFile);
    int getNameRank(const std::string& name) const;
    int getAbsoluteNumber(const std::string& name) const;
private:
    std::map<std::string, int> mNames;
    bool nameExistsAndIncrement(const std::string& name);
    void addNewName(const std::string& name);
};
```

Here are the new method implementations:

```
// Reads the names from the file and populates the database.
// The database is a map associating names with their frequency.
NameDB::NameDB(const string& nameFile)
{
    // Open the file and check for errors.
    ifstream inFile(nameFile.c_str());
    if (!inFile) {
        throw invalid_argument("Unable to open file");
    }
    // Read the names one at a time.
    string name;
    while (inFile >> name) {
        // Look up the name in the database so far.
        if (!nameExistsAndIncrement(name)) {
            // If the name exists in the database, the
            // method incremented it, so we just continue.
            // We get here if it didn't exist, in which case
            // we add it with a count of 1.
            addNewName(name);
        }
    }
}
```

```
        }
        inFile.close();
    }

// Returns true if the name exists in the database, false
// otherwise. If it finds it, it increments it.
bool NameDB::nameExistsAndIncrement(const string& name)
{
    // Find the name in the map.
    auto res = mNames.find(name);
    if (res != end(mNames)) {
        res->second++;
        return true;
    }
    return false;
}

// Adds a new name to the database.
void NameDB::addNewName(const string& name)
{
    mNames[name] = 1;
}

// Returns the rank of the name.
// First looks up the name to obtain the number of babies with that name.
// Then iterates through all the names, counting all the names with a higher
// count than the specified name. Returns that count as the rank.
int NameDB::getNameRank(const string& name) const
{
    int num = getAbsoluteNumber(name);
    // Check if we found the name.
    if (num == -1) {
        return -1;
    }
    // Now count all the names in the map that have
    // count higher than this one. If no name has a higher count,
    // this name is rank number 1. Every name with a higher count
    // decreases the rank of this name by 1.
    int rank = 1;
    for (auto& entry : mNames) {
        if (entry.second > num) {
            rank++;
        }
    }
    return rank;
}

// Returns the count associated with this name.
int NameDB::getAbsoluteNumber(const string& name) const
{
    auto res = mNames.find(name);
    if (res != end(mNames)) {
        return res->second;
    }
    return -1;
}
```

Profile of the Second Attempt

By following the same steps shown earlier, you can obtain the gprof performance data on the new version of the program. The data are quite encouraging:

index	%time	self	children	called	name
[1]	100.0	0.00	0.21		main [1]
		0.02	0.18	1/1	NameDB::NameDB [2]
		0.00	0.01	1/1	NameDB::~NameDB [13]
		0.00	0.00	3/3	NameDB::getNameRank [28]
[2]	95.2	0.02	0.18	1	NameDB::NameDB [2]
		0.02	0.16	500500/500500	NameDB::nameExistsAndIncrement [3]
		0.00	0.00	1000/1000	NameDB::addNewName [24]
		0.00	0.00	1/1	map::map [87]

If you run this on your machine, the output will be different. It's even possible that you will not see the data for `NameDB` methods in your output. Because of the efficiency of this second attempt, the timings are getting so small that you might see more `map` methods in the output than `NameDB` methods.

On my test system, `main()` now takes only 0.21 seconds: a 67-fold improvement! There are certainly further improvements that you could make on this program. For example, the current constructor performs a lookup to see if the name is already in the `map`, and if not, adds it to the `map`. You could combine these two operations. You could always use the `insert()` method of the `map` without first checking if the name already exists. The `insert()` method returns a `pair<iterator, bool>`. The Boolean will be `true` if it added the name to the `map`, and `false` if the name was already present. When the name is already in the `map`, you increment its count, which you can reach through the iterator in the returned pair. This iterator is a `map` entry, so to access the count, you use `res.first->second`. To implement this improvement, you can remove the `nameExistsAndIncrement()` and `addNewName()` methods, and change the constructor as follows:

```
NameDB::NameDB(const string& nameFile)
{
    // Open the file and check for errors
    ifstream inFile(nameFile.c_str());
    if (!inFile) {
        throw invalid_argument("Unable to open file");
    }
    // Read the names one at a time.
    string name;
    while (inFile >> name) {
        auto res = mNames.insert(make_pair(name, 1));
        if (res.second == false) {
            res.first->second++; // Already in the map, increment count
        }
    }
    inFile.close();
}
```

`getNameRank()` still uses a loop that iterates over all elements in the `map`. A good exercise for the reader is to come up with another data structure so that the linear iteration in `getNameRank()` can be avoided.

Profiling Example with Visual C++ 2013

Most editions of Microsoft Visual C++ 2013 come with a great built-in profiler, which is briefly discussed in this section. The VC++ profiler has a complete graphical user interface. This book does not recommend one or the other profiler, but it is always good to have an idea of what a command-line based profiler like gprof can provide in comparison to a GUI-based profiler like the one included with VC++.

Profile of the First Design Attempt

To start profiling an application in Visual C++ 2013, you first need to open the project in Visual Studio. This example uses the same NameDB code as in the first inefficient design attempt earlier. This code is not repeated here. Once your project is opened in Visual Studio, click on the “Analyze” menu and then choose “Performance and Diagnostics.” A new window appears. Figure 25-1 shows a screenshot of how this window might look like.

In this new window, enable the “Performance Wizard” option and click the “Start” button. This will start a wizard. The first page of this wizard is shown in Figure 25-2.

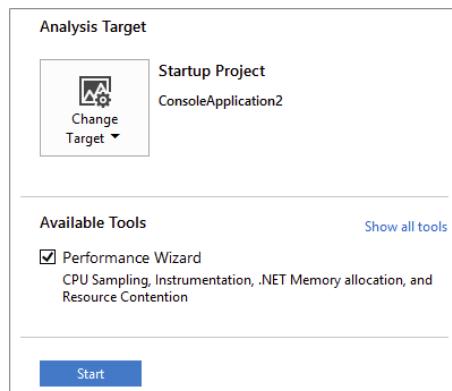


FIGURE 25-1

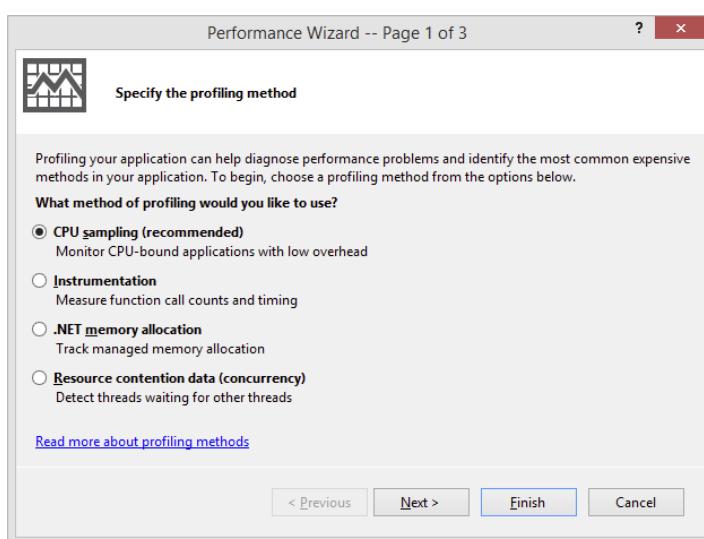


FIGURE 25-2

Depending on the version of VC++ 2013, there are a number of different profiling methods available. The following non-exhaustive list explains three of them:

- **CPU Sampling:** Used to monitor applications with low overhead. This means that the act of profiling the application will not have a big performance impact on the target application.

- **Instrumentation:** This method will add extra code to the application to be able to accurately count the number of function calls and to time individual function calls. However, this method has a much bigger performance impact on the application. It is recommended to use the CPU Sampling method first to get an idea about the bottlenecks in your application. If this method does not give you enough information, you can try the Instrumentation method.
- **Concurrency:** This allows you to monitor multithreaded applications. It allows you to graphically see which threads are running, which threads are waiting for something, and so on.

For this profiling example, leave the default CPU Sampling method and click the “Next” button. The next page of the wizard asks you to select the application that you want to profile. Here you should select your NameDB project and click the “Next” button. On the last page of the wizard you can enable the “Launch profiling after the wizard finishes” and then click the “Finish” button. It is possible that you will get a message saying that you don’t have the right credentials for profiling, and whether you would like to upgrade your credentials. If you get this message, you should allow VC++ to upgrade your credentials otherwise the profiler will not work.

When the program execution is finished, Visual Studio automatically opens the profiling report. Figure 25-3 shows how this report might look when profiling the first attempt of the NameDB application.

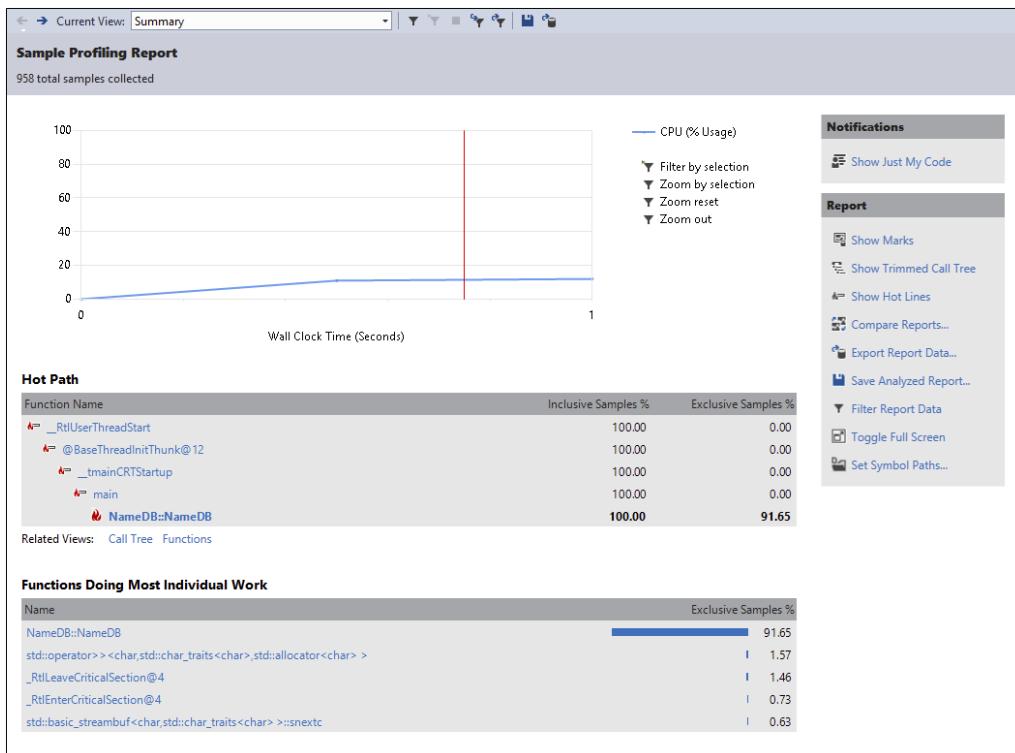


FIGURE 25-3

From this report you can immediately see the hot path. Just like with gprof, it shows that the `NameDB` constructor takes up most of the running time of the program. The Visual Studio profiling report is interactive. For example, you can drill down the `NameDB` constructor by clicking on it. This results in a drill-down report for that function, which looks like Figure 25-4.

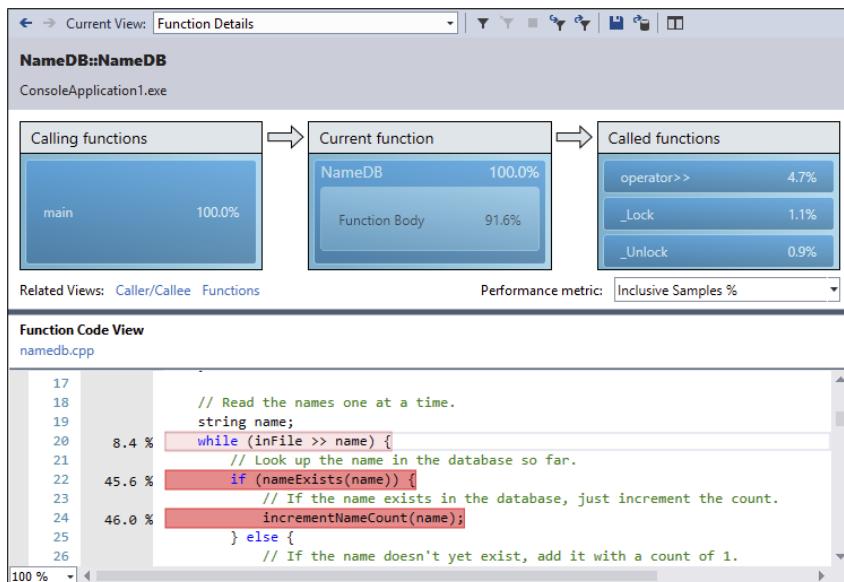


FIGURE 25-4

This drill-down view shows a graphical breakdown at the top, and the actual code of the method at the bottom. The code view shows the percentage of the running time that a line needed. The line using up most of the time is shown in red. When you are interactively browsing the profiling report, you can always go back by using the back arrow at the top-left of the report.

At the top of the report there is also a drop-down menu, which you can use to quickly jump to certain summary or details pages. If you go back to the “Summary” of the profiling report, you can see that there is a “Show Trimmed Call Tree” link on the right. Clicking that link displays a trimmed call tree showing you an alternative view of the hot path in your code. This is shown in Figure 25-5.

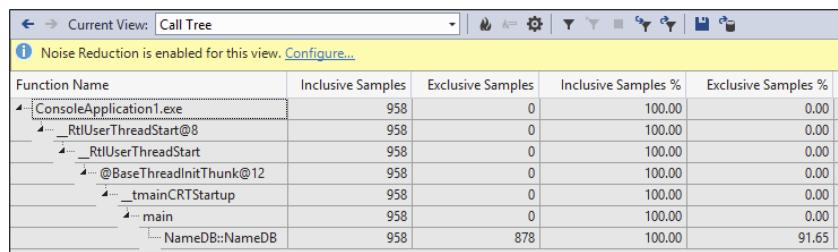


FIGURE 25-5

Also in this view, you immediately see that `main()` is calling the `NameDB` constructor, which is using up most of the time.

SUMMARY

This chapter discussed the key aspects of efficiency and performance in C++ programs, and provided several specific tips and techniques for designing and writing more efficient applications. Hopefully you gained an appreciation for the importance of performance and for the power of profiling tools.

There are two important things to remember from this chapter. The first thing is that you should not get too obsessed with performance while designing and coding. It's recommended to first make a correct, well-structured design and implementation, then use a profiler, and only optimize those parts that are flagged by a profiler as being a performance bottleneck. The second thing to remember is that design-level efficiency is far more important than language-level efficiency. For example, don't use algorithms or data structures with bad complexity if there are better ones available.

26

Conquering Debugging

WHAT'S IN THIS CHAPTER?

- The Fundamental Law of Debugging and bug taxonomies
- Tips for avoiding bugs
- How to plan for bugs
- The different kinds of memory errors
- How to use a debugger to pinpoint code causing a bug

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at www.wrox.com/go/proc++3e on the Download Code tab.

Your code will contain bugs. Every professional programmer would like to write bug-free code, but the reality is that few software engineers succeed in this endeavor. As computer users know, bugs are endemic in computer software. The software that you write is probably no exception. Therefore, unless you plan to bribe your co-workers into fixing all your bugs, you cannot be a professional C++ programmer without knowing how to debug C++ code. One factor that often distinguishes experienced programmers from novices is their debugging skills.

Despite the obvious importance of debugging, it is rarely given enough attention in courses and books. Debugging seems to be the type of skill that everyone wants you to know, but no one knows how to teach. This chapter attempts to provide concrete debugging guidelines and techniques.

This chapter includes an introduction to the Fundamental Law of Debugging and bug taxonomies, followed by tips for avoiding bugs. Techniques for planning for bugs include error logging, debug traces, assertions, and crash dump. Specific tips are given for debugging

the problems that arise, including techniques for reproducing bugs, debugging reproducible bugs, debugging nonreproducible bugs, debugging memory errors, and debugging multithreaded programs. The chapter concludes with a step-by-step debugging example.

THE FUNDAMENTAL LAW OF DEBUGGING

The first rule of debugging is to be honest with yourself and admit that your program will contain bugs. This realistic assessment enables you to put your best effort into preventing bugs from crawling into your program in the first place while you simultaneously include the necessary features to make debugging as easy as possible.

WARNING *The Fundamental Law of Debugging: Avoid bugs when you’re coding, but plan for bugs in your code.*

BUG TAXONOMIES

A *bug* in a computer program is incorrect run-time behavior. This undesirable behavior includes both *catastrophic* and *noncatastrophic bugs*. Examples of catastrophic bugs are program death, data corruption, operating system failures, or some other horrific outcome. A catastrophic bug can also manifest itself external to the software or computer system running the software; for example, medical software might contain a catastrophic bug causing a massive radiation overdose to a patient. Noncatastrophic bugs are bugs that cause the program to behave incorrectly in more subtle ways; for example, a web browser might return the wrong web page, or a spreadsheet application might calculate the standard deviation of a column incorrectly.

There are also so-called *cosmetic bugs*, where something is visually not correct, but otherwise works correctly. For example, a button in a user interface is left-enabled when it shouldn’t be, but clicking it does nothing. All computations are perfectly correct, the program does not crash, but it doesn’t look as “nice” as it should.

The underlying cause, or *root cause*, of the bug is the mistake in the program that causes this incorrect behavior. The process of debugging a program includes both determining the root cause of the bug and fixing the code so that the bug will not occur again.

AVOIDING BUGS

It’s impossible to write completely bug-free code, so debugging skills are important. However, a few tips can help you to minimize the number of bugs:

- **Read this book from cover to cover:** Learn the C++ language intimately, especially pointers and memory management. Then, recommend this book to your friends and coworkers so they avoid bugs too.

- **Design *before* you code:** Designing *while* you code tends to lead to convoluted designs that are harder to understand and are more error-prone. It also makes you more likely to omit possible edge cases and error conditions.
- **Do code reviews:** If you wrote a more complicated or risky piece of code, ask a co-worker to review your code. Sometimes it takes a fresh perspective to notice problems.
- **Test, test, and test again:** Thoroughly test your code, and have *others* test your code! They are more likely to find problems you haven't thought of.
- **Write automated unit tests:** Unit tests are designed to test isolated functionality. You should write unit tests for all implemented features. Run these unit tests automatically as part of your continuous integration setup.
- **Expect error conditions, and handle them appropriately:** In particular, plan for and handle out-of-memory conditions. They will occur. See Chapter 13.
- **Use smart pointers to avoid memory leaks:** Smart pointers automatically free resources when they are not needed anymore.
- **Don't ignore compiler warnings:** Configure your compiler to compile with a high warning level. Do not blindly ignore warnings. Ideally, you should enable an option in your compiler to treat warnings as errors. This forces you to address each warning.
- **Use static code analysis:** A static code analyzer helps you to pinpoint problems in your code by analyzing your source code.
- **Use good coding style:** Strive for readability and clarity, add code comments (not only interface comments), use the `override` keyword, and so on. This makes it easier for other people to understand your code.

PLANNING FOR BUGS

Your programs should contain features that enable easier debugging when the inevitable bugs arise. This section describes these features and presents sample implementations, where appropriate, that you can incorporate into your own programs.

Error Logging

Imagine this scenario: You have just released a new version of your flagship product, and one of the first users reports that the program “stopped working.” You attempt to pry more information from the user, and eventually discover that the program died in the middle of an operation. The user can’t quite remember what he was doing, or if there were any error messages. How will you debug this problem?

Now imagine the same scenario, but in addition to the limited information from the user, you are also able to examine the error log on the user’s computer. In the log you see a message from your program that says “Error: unable to allocate memory.” Looking at the code near the spot where that

error message was generated, you find a line in which you dereferenced a pointer without checking for `nullptr`. You've found the root cause of your bug!

Error logging is the process of writing error messages to persistent storage so that they will be available following an application, or even machine, death. Despite the example scenario, you might still have doubts about this strategy. Won't it be obvious by your program's behavior if it encounters errors? Won't the user notice if something goes wrong? As the preceding example shows, user reports are not always accurate or complete. In addition, many programs, such as the operating system kernel and long-running daemons like `inetd` or `syslogd` on Unix, are not interactive and run unattended on a machine. The only way these programs can communicate with users is through error logging. In many cases, a program might also want to automatically recover from certain errors, and hide the error from the user. Still, having logs of those errors available can be invaluable to improve the overall stability of the program.

Thus, your program should log errors as it encounters them. That way, if a user reports a bug, you will be able to examine the log files on the machine to see if your program reported any errors prior to encountering the bug. Unfortunately, error logging is platform dependent: C++ does not contain a standard logging mechanism. Examples of platform-specific logging mechanisms include the `syslog` facility in Unix and the event reporting API in Windows. You should consult the documentation for your development platform. There are also some open-source implementations of cross-platform logging classes. One example is `Boost.Log`, available at <http://www.boost.org/>.

Now that you're convinced that error logging is a great feature to add to your programs, you might be tempted to log error messages every few lines in your code, so that, in the event of any bug, you'll be able to trace the code path that was executing. These types of error messages are appropriately called "traces."

However, you should not write these traces to error logs for two reasons. First, writing to persistent storage is slow. Even on systems that write the logs asynchronously, logging that much information will slow down your program. Second, and most important, most of the information that you would put in your traces is not appropriate for the end user to see. It will just confuse the user, leading to unwarranted service calls. That said, tracing is an important debugging technique under the correct circumstances, as described in the next section.

Here are some specific guidelines for the types of errors your programs should log:

- Unrecoverable errors, such as an inability to allocate memory or a system call failing unexpectedly.
- Errors for which an administrator can take action, such as low memory, an incorrectly formatted data file, an inability to write to disk, or a network connection being down.
- Unexpected errors such as a code path that you never expected to take or variables with unexpected values. Note that your code should "expect" users to enter invalid data and should handle it appropriately. An unexpected error would represent a bug in your program.
- Potential security breaches such as a network connection attempted from an unauthorized address, or too many network connections attempted (denial of service).

It is also useful to log warnings, or recoverable errors, which allow you to investigate if you can possibly avoid them.

Additionally, most APIs allow you to specify a *log level* or *error level*, typically error, warning, and info. You can log non-error conditions under a log level that is less severe than “error.” For example, you might want to log significant state changes in your application, or startup and shutdown of the program. You also might consider giving your users a way to adjust the log level of your program at run time so that they can customize the amount of logging that occurs.

Debug Traces

When debugging complicated problems, public error messages generally do not contain enough information. You often need a complete trace of the code path taken, or values of variables before the bug showed up. In addition to basic messages, it’s sometimes helpful to include the following information in debug traces:

- The thread ID, if it’s a multithreaded program
- The name of the function that generates the trace
- The name of the source file in which the code that generates the trace lives

You can add this tracing to your program through a special debug mode, or via a *ring buffer*. Both of these methods are explained in detail in the next sections. Note that in multithreaded programs you have to make your trace logging thread-safe. See Chapter 23 for details on multithreaded programming.

NOTE *Be careful with logging too much detail. You don’t want to leak intellectual property through your log files.*

Debug Mode

The first technique to add debug traces is to provide a debug mode for your program. In debug mode, the program writes trace output to standard error or to a file, and perhaps does extra checking during execution. There are several ways to add a debug mode to your program.

Start-Time Debug Mode

Start-time debug mode allows your application to run with or without debug mode depending on a command-line argument. This strategy includes the debug code in the “release” binary, and allows debug mode to be enabled at a customer site. However, it does require users to restart the program in order to run it in debug mode, which may prevent you from obtaining useful information about certain bugs.

The following example is a simple program implementing a start-time debug mode. This program doesn’t do anything useful; it is only for demonstrating the technique.

All logging functionality is wrapped in a `Logger` class. This class has two `static` data members: The name of the log file, and a Boolean saying whether or not logging is enabled or disabled. The class has a `static` public `log()` variadic template method. Variadic templates are discussed in Chapter 21.

Note that the log file is opened, flushed, and closed on each call to `log()`. This might lower performance a bit; however, it does guarantee correct logging, which is more important.

```

class Logger
{
public:
    static void enableLogging(bool enable) { msLoggingEnabled = enable; }
    static bool isLoggingEnabled() { return msLoggingEnabled; }

    template<typename... Args>
    static void log(const Args&... args)
    {
        if (!msLoggingEnabled)
            return;

        ofstream ofs(msDebugFileName, ios_base::app);
        if (ofs.fail()) {
            cerr << "Unable to open debug file!" << endl;
            return;
        }
        logHelper(ofs, args...);
        ofs << endl;
    }

private:
    template<typename T1>
    static void logHelper(ofstream& ofs, const T1& t1)
    {
        ofs << t1;
    }

    template<typename T1, typename... Tn>
    static void logHelper(ofstream& ofs, const T1& t1, const Tn&... args)
    {
        ofs << t1;
        logHelper(ofs, args...);
    }

    static const char* msDebugFileName;
    static bool msLoggingEnabled;
};

const char* Logger::msDebugFileName = "debugfile.out";
bool Logger::msLoggingEnabled = false;

```

The following helper macro is defined to make it easy to log something. It uses `__func__`, defined by the C++ standard, a predefined variable that contains the name of the current function. If your compiler doesn't support this standard `__func__` yet, you should check its documentation. Maybe it supports `__FUNCTION__` instead, or something similar.

```
#define log(...) Logger::log(__func__, "() : ", __VA_ARGS__)
```

This macro replaces every call to `log()` in your code with a call to `Logger::log()`. The macro automatically includes the function name as first argument to `Logger::log()`. For example:

```
log("given argument: ", *obj);
```

The `log()` macro replaces this with:

```
Logger::log(__func__, "() : ", "given argument: ", *obj);
```

Start-time debug mode needs to parse the command-line arguments to find out if it should enable debug mode or not. Unfortunately, there is no standard library functionality in C++ for parsing command-line arguments. This program uses a simple `isDebugSet()` function to check for the debug flag among all the command-line arguments, but a function to parse all command-line arguments would need to be more sophisticated.

```
bool isDebugSet(int argc, char* argv[])
{
    for (int i = 0; i < argc; i++) {
        if (strcmp(argv[i], "-d") == 0) {
            return true;
        }
    }
    return false;
}
```

Some arbitrary test code is used to exercise the debug mode in this example. Two classes are defined, `ComplicatedClass` and `UserCommand`. Both classes define an `operator<<` to write instances of them to a stream, because the `Logger` class uses this operator to dump objects to the log.

```
class ComplicatedClass
{
public:
    ComplicatedClass() {}
};

ostream& operator<<(ostream& ostr, const ComplicatedClass& src)
{
    ostr << "ComplicatedClass";
    return ostr;
}

class UserCommand
{
public:
    UserCommand() {}
};

ostream& operator<<(ostream& ostr, const UserCommand& src)
{
    ostr << "UserCommand";
    return ostr;
}
```

Here is some test code with a number of log calls:

```
UserCommand getNextCommand(ComplicatedClass* obj)
{
    UserCommand cmd;
    return cmd;
}
```

```

void processUserCommand(UserCommand& cmd)
{
    // details omitted for brevity
}

void trickyFunction(ComplicatedClass* obj)
{
    log("given argument: ", *obj);

    for (size_t i = 0; i < 100; ++i) {
        UserCommand cmd = getNextCommand(obj);
        log("retrieved cmd ", i, ":", cmd);

        try {
            processUserCommand(cmd);
        } catch (const exception& e) {
            log("received exception from processUserCommand(): ", e.what());
        }
    }
}

int main(int argc, char* argv[])
{
    Logger::enableLogging(isDebugSet(argc, argv));

    if (Logger::isLoggingEnabled()) {
        // Print the command-line arguments to the trace
        for (int i = 0; i < argc; i++) {
            log(argv[i]);
        }
    }

    ComplicatedClass obj;
    trickyFunction(&obj);
    // Rest of the function not shown
    return 0;
}

```

There are two ways to run this application:

```

> STDebug
> STDebug -d

```

Debug mode is activated only when the `-d` argument is specified on the command line.

WARNING *Macros in C++ should be avoided as much as possible because they can be hard to debug. However, for logging purposes, using a simple macro is acceptable and it makes using the logging code much easier.*

Compile-Time Debug Mode

Instead of enabling or disabling debug mode through a command-line argument, you could also use a preprocessor symbol such as `DEBUG_MODE` and `#ifdefs` to selectively compile the debug code into

your program. In order to generate a debug version of this program, you would have to compile it with the symbol `DEBUG_MODE` defined. Your compiler should allow you to define symbols during compilation; consult your compiler's documentation for details. For example, GCC allows you to specify `-Dsymbol` through the command-line; and Microsoft VC++ allows you to specify the symbols through the Visual Studio IDE, or if you use the VC++ command-line, `/D symbol`.

The advantage of this method is that your debug code is not compiled into the “release” binary, and so does not increase its size. The disadvantage is that there is no way to enable debugging at a customer site for testing or following the discovery of a bug.

An example implementation is given in `CTDebug.cpp` in the downloadable source code archive.

Run-Time Debug Mode

The most flexible way to provide a debug mode is to allow it to be enabled or disabled at run time. One way to provide this feature is to supply an asynchronous interface that controls debug mode on the fly. In a GUI program, this interface could take the form of a menu command. In a CLI (Command Line Interface) program, this interface could be an asynchronous command that makes an interprocess call into the program (using sockets, signals, or remote procedure calls, for example). C++ provides no standard way to perform interprocess communication or GUIs, so an example of this technique is not shown.

Ring Buffers

Debug mode is useful for debugging reproducible problems and for running tests. However, bugs often appear when the program is running in non-debug mode, and by the time you or the customer enables debug mode, it is too late to gain any information about the bug. One solution to this problem is to enable tracing in your program at all times. You usually need only the most recent traces to debug a program, so you should store only the most recent traces at any point in a program's execution. One way to provide this limitation is through careful use of log file rotations.

However, for performance reasons, it is better that your program doesn't log these traces continuously to disk. Instead, it should store them in memory and provide a mechanism to dump all the trace messages to standard error or to a log file if the need arises.

A common technique is to use a *ring buffer* to store a fixed number of messages, or messages in a fixed amount of memory. When the buffer fills up, it starts writing messages at the beginning of the buffer again, overwriting the older messages. This cycle can repeat indefinitely. The following sections provide an implementation of a ring buffer and show you how you can use it in your programs.

Ring Buffer Interface

The following `RingBuffer` class provides a simple debug ring buffer. The client specifies the number of entries in the constructor and adds messages with the `addEntry()` method. Once the number of entries exceeds the number allowed, new entries overwrite the oldest entries in the buffer. The buffer also provides the option to output entries to a stream as they are added to the buffer. The client can specify an output stream in the constructor, and can reset it with the `setOutput()` method. Finally, the `operator<<` streams the entire buffer to an output stream. This implementation uses a variadic template method. Variadic templates are discussed in Chapter 21.

```

class RingBuffer
{
public:
    // Constructs a ring buffer with space for numEntries.
    // Entries are written to *ostr as they are queued (optional).
    RingBuffer(size_t numEntries = kDefaultNumEntries,
               std::ostream* ostr = nullptr);
    virtual ~RingBuffer();

    // Adds the string to the ring buffer, possibly overwriting the
    // oldest string in the buffer (if the buffer is full).
    template<typename... Args>
    void addEntry(const Args&... args)
    {
        std::ostringstream os;
        addEntryHelper(os, args...);
        addStringEntry(os.str());
    }

    // Streams the buffer entries, separated by newlines, to ostr.
    friend std::ostream& operator<<(std::ostream& ostr, RingBuffer& rb);

    // Sets the output stream to which entries are streamed as they are added.
    // Returns the old output stream.
    std::ostream* setOutput(std::ostream* newOstr);

private:
    std::vector<std::string> mEntries;
    std::vector<std::string>::iterator mNext;
    std::ostream* mOstr;
    bool mWrapped;
    static const size_t kDefaultNumEntries = 500;

    template<typename T1>
    void addEntryHelper(std::ostringstream& os, const T1& t1)
    {
        os << t1;
    }

    template<typename T1, typename... Tn>
    void addEntryHelper(std::ostringstream& os, const T1& t1,
                       const Tn&... args)
    {
        os << t1;
        addEntryHelper(os, args...);
    }

    void addStringEntry(std::string&& entry);
};

```

Ring Buffer Implementation

This implementation of the ring buffer stores a fixed number of `string` objects. This approach certainly is not the most efficient solution. Other possibilities would be to provide a fixed number

of bytes of memory for the buffer. However, this implementation should be sufficient unless you're writing a high-performance application.

For multithreaded programs it's useful to add the ID of the thread and a timestamp to each trace entry. Of course, the ring buffer has to be made thread-safe before using it in a multithreaded application. See Chapter 23 for multithreaded programming.

```

// Initialize the vector to hold exactly numEntries. The vector size
// does not need to change during the lifetime of the object.
// Initialize the other members.
RingBuffer::RingBuffer(size_t numEntries, ostream* ostr) : mEntries(numEntries),
    mNext(begin(mEntries)), mOstr(ostr), mWrapped(false)
{
}

RingBuffer::~RingBuffer()
{
}

// The addEntry algorithm is pretty simple: add the entry to the next
// free spot, then reset mNext to indicate the free spot after
// that. If mNext reaches the end of the vector, it starts over at 0.
//
// The buffer needs to know if the buffer has wrapped or not so
// that it knows whether to print the entries past mNext in operator<<
void RingBuffer::addStringEntry(string&& entry)
{
    // If there is a valid ostream, write this entry to it.
    if (mOstr) {
        *mOstr << entry << endl;
    }

    // Move the entry to the next free spot and increment
    // mNext to point to the free spot after that.
    *mNext = std::move(entry);
    ++mNext;

    // Check if we've reached the end of the buffer. If so, we need to wrap.
    if (mNext == end(mEntries)) {
        mNext = begin(mEntries);
        mWrapped = true;
    }
}

// Set the output stream.
ostream* RingBuffer::setOutput(ostream* newOstr)
{
    ostream* ret = mOstr;
    mOstr = newOstr;
    return ret;
}

// operator<< uses an ostream_iterator to "copy" entries directly
// from the vector to the output stream.
//

```

```

// operator<< must print the entries in order. If the buffer has wrapped,
// the earliest entry is one past the most recent entry, which is the entry
// indicated by mNext. So first print from entry mNext to the end.
//
// Then (even if the buffer hasn't wrapped) print from the beginning to mNext-1.
ostream& operator<<(ostream& ostr, RingBuffer& rb)
{
    if (rb.mWrapped) {
        // If the buffer has wrapped, print the elements from
        // the earliest entry to the end.
        copy(rb.mNext, end(rb.mEntries), ostream_iterator<string>(ostr, "\n"));
    }

    // Now, print up to the most recent entry.
    // Go up to mNext because the range is not inclusive on the right side.
    copy(begin(rb.mEntries), rb.mNext, ostream_iterator<string>(ostr, "\n"));
    return ostr;
}

```

Using the Ring Buffer

In order to use the ring buffer, you can create an instance of it and start adding messages to it. When you want to print the buffer, just use `operator<<` to print it to the appropriate `ostream`. Here is the earlier start-time debug mode program modified to show use of a ring buffer instead. Changes are highlighted. The definitions of the `ComplicatedClass` and `UserCommand` classes, and the functions `getNextCommand()` and `processUserCommand()` are not shown. They are identical as before.

```

RingBuffer debugBuf;

void trickyFunction(ComplicatedClass* obj)
{
    // Trace log the values with which this function starts.
    debugBuf.addEntry(__func__, "(): given argument: ", *obj);

    for (size_t i = 0; i < 100; ++i) {
        UserCommand cmd = getNextCommand(obj);
        debugBuf.addEntry(__func__, "(): retrieved cmd ", i, ": ", cmd);

        try {
            processUserCommand(cmd);
        } catch (const exception& e) {
            debugBuf.addEntry(__func__,
                "(): received exception from processUserCommand():", e.what());
        }
    }
}

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

```

```

{
    // Print the command-line arguments
    for (int i = 0; i < argc; i++) {
        debugBuf.addEntry(argv[i]);
    }

    ComplicatedClass obj;
    trickyFunction(&obj);

    // Print the current contents of the debug buffer to cout
    cout << debugBuf;

    return 0;
}

```

Displaying the Ring Buffer Contents

Storing trace debug messages in memory is a great start, but in order for them to be useful, you need a way to access these traces for debugging.

Your program should provide a “hook” to tell it to export the messages. This hook could be similar to the interface you would use to enable debugging at run time. Additionally, if your program encounters a fatal error that causes it to exit, it could export the ring buffer automatically to a log file before exiting.

Another way to retrieve these messages is to obtain a memory dump of the program. Each platform handles memory dumps differently, so you should consult a book or expert on your platform.

Assertions

The `<cassert>` header defines an `assert` macro. It takes a Boolean expression and, if the expression evaluates to `false`, prints an error message and terminates the program. If the expression evaluates to `true`, it does nothing.

WARNING *Normally, you should avoid any library functions or macros that can terminate your program. The assert macro is an exception. If an assertion triggers, it means that some assumption is wrong or that something is catastrophically, unrecoverably wrong, and the only sane thing to do is to terminate the application at that very moment, instead of continuing.*

Assertions allow you to “force” your program to exhibit a bug at the exact point where that bug originates. If you didn’t assert at that point, your program might proceed with those incorrect values, and the bug might not show up until much later. Thus, assertions allow you to detect bugs earlier than you otherwise would.

NOTE *The behavior of the standard assert macro depends on the `NDEBUG` pre-processor symbol: If the symbol is not defined, the assertion takes place, otherwise it is ignored. Compilers often define this symbol when compiling “release” builds. If you want to leave assertions in release builds, you may have to change your compiler settings, or write your own version of assert that isn’t affected by the value of `NDEBUG`.*

You could use assertions in your code whenever you are “assuming” something about the state of your variables. For example, if you call a library function that is supposed to return a pointer and claims never to return `nullptr`, throw in an `assert` after the function call to make sure that the pointer isn’t `nullptr`.

Note that you should assume as little as possible. For example, if you are writing a library function, don’t assert that the parameters are valid. Instead, check the parameters and return an error code or throw an exception if they are invalid.

As a rule, assertions should only be used for cases that are truly problematic, and should therefore never be ignored when occurring during development. If you hit an assertion during development, fix it, don’t just disable the assertion.

WARNING *Be careful not to put any code that must be executed for correct program functioning inside assertions. For example, a line like this is probably asking for trouble: `assert(myFunctionCall() != nullptr)`. If a release build of your code strips assertions, then the call to `myFunctionCall()` is stripped as well.*

Static Assertions

The assertions discussed in the previous section are evaluated at run time. `static_assert` allows assertions evaluated at compile time. A `static_assert` requires two parameters: an expression to evaluate and a string. When the expression evaluates to `false`, the compiler issues an error that contains the given string. A simple example is to check `INT_MAX`:

```
static_assert(INT_MAX >= 0x7FFFFFFF,
    "Code requires INT_MAX to be at least 0x7FFFFFFF.");
```

If you compile this with a compiler where `INT_MAX` is less than `0x7FFFFFFF`, the compiler issues an error that could look as follows:

```
test.cpp(3): error C2338: Code requires INT_MAX to be at least 0x7FFFFFFF.
```

Another example where `static_asserts` are pretty powerful is in combination with type traits. Type traits are discussed in Chapter 21. For example, if you write a function template or class template, you could use `static_asserts` together with type traits to issue compiler errors when template

types don't satisfy certain conditions. The following example requires that the template type for `process()` has `Base1` as its base class:

```
#include <type_traits>
using namespace std;
class Base1 {};
class Base1Child : public Base1 {};
class Base2 {};
class Base2Child : public Base2 {};

template<typename T>
void process(const T& t)
{
    static_assert(is_base_of<Base1, T>::value, "Base1 should be a base for T.");
}

int main()
{
    process(Base1());
    process(Base1Child());
    //process(Base2());           // Error
    //process(Base2Child()); // Error
    return 0;
}
```

If you try to call `process()` with an instance of `Base2` or `Base2Child`, the compiler issues an error that could look as follows:

```
test.cpp(13): error C2338: Base1 should be a base for T.
              test.cpp(21) : see reference to function template
                           instantiation 'void process<Base2>(const T &)' being compiled
                           with
                           [
                           T=Base2
                           ]
```

Crash Dumps

Make sure your program creates *crash dumps*, also called *memory dumps*, *core dumps*, and so on. How you create such dumps is platform dependent, so you should consult the documentation of your platform.

Also make sure you set up a *symbol server* and a *source code server*. The symbol server is used to store debugging symbols of released binaries of your software. These symbols are used later on to interpret crash dumps from customers. The source code server, discussed in Chapter 24, stores all revisions of your source code. When debugging crash dumps, this source code server is used to download the correct source code for the revision of your software that created the crash dump.

The exact procedure of analyzing crash dumps depends on your platform and compiler, so consult their documentation.

From personal experience: A crash dump is often worth more than a thousand bug reports.

DEBUGGING TECHNIQUES

Debugging a program can be incredibly frustrating. However, with a systematic approach it becomes significantly easier. Your first step in trying to debug a program should always be to reproduce the bug. Depending on whether or not you can reproduce the bug, your subsequent approach will differ. The next four sections explain how to reproduce bugs, how to debug reproducible bugs, how to debug nonreproducible bugs, and how to debug regressions. Additional sections explain details about debugging memory errors and debugging multithreaded programs.

Reproducing Bugs

If you can reproduce the bug consistently, it will be much easier to determine the root cause. Finding the root cause of bugs that are not reproducible is difficult, if not impossible.

As a first step to reproduce the bug, run the program with exactly the same inputs as the run when the bug first appeared. Be sure to include all inputs, from the program’s startup to the time of the bug’s appearance. A common mistake is to attempt to reproduce the bug by performing only the triggering action. This technique may not reproduce the bug because the bug might be caused by an entire sequence of actions.

For example, if your web browser program dies when you request a certain web page, it may be due to memory corruption triggered by that particular request’s network address. On the other hand, it may be because your program records all requests in a queue, with space for one million entries, and this entry was number one million and one. Starting the program over and sending one request certainly wouldn’t trigger the bug in that case.

Sometimes it is impossible to emulate the entire sequence of events that leads to the bug. Perhaps the bug was reported by someone who can’t remember everything that he or she did. Alternatively, maybe the program was running for too long to emulate every input. In that case, do your best to reproduce the bug. It takes some guesswork, and can be time-consuming, but effort at this point will save time later in the debugging process. Here are some techniques you can try:

- Repeat the triggering action in the correct environment and with as many inputs as possible similar to the initial report.
- Do a quick review of the code related to the bug. More often than not, you’ll find a likely cause that will guide you in reproducing the problem.
- Run automated tests that exercise similar functionality. Reproducing bugs is one benefit of automated tests. If it takes 24 hours of testing before the bug shows up, it’s preferable to let those tests run on their own rather than spend 24 hours of your time trying to reproduce it.
- If you have the necessary hardware available, running slight variations of tests concurrently on different machines can sometimes save time.
- Run stress tests that exercise similar functionality. If your program is a web server that died on a particular request, try running as many browsers as possible simultaneously that make that request.

After you are able to reproduce the bug consistently, you should attempt to find the smallest sequence that triggers the bug. You can start with the minimum sequence, containing only the

triggering action, and slowly expand the sequence to cover the entire sequence from startup until the bug is triggered. This will result in the simplest and most efficient test case to reproduce it, which makes it simpler to find the root cause of the problem, and it's easier to verify the fix.

Debugging Reproducible Bugs

When you can reproduce a bug consistently and efficiently, it's time to figure out the problem in the code that causes the bug. Your goal at this point is to find the exact lines of code that trigger the problem. You can use two different strategies:

1. **Logging debug messages:** By adding enough debug messages to your program and watching its output when you reproduce the bug, you should be able to pinpoint the exact lines of code where the bug occurs. If you have a debugger at your disposal, adding debug messages is usually not recommended because it requires modifications to the program and can be time-consuming. However, if you have already instrumented your program with debug messages as described earlier, you might be able to find the root cause of your bug by running your program in debug mode while reproducing the bug. Note that bugs sometimes disappear simply by enabling logging because the act of enabling logging potentially changes the timings of your application slightly.
2. **Using a debugger:** Debuggers allow you to step through the execution of your program and to view the state of memory and the values of variables at various points. They are often indispensable tools for finding the root cause of bugs. When you have access to the source code, you will use a *symbolic debugger*: a debugger that utilizes the variable names, class names, and other symbols in your code. In order to use a symbolic debugger you must instruct your compiler to generate debug symbols.

The debugging example at the end of this chapter demonstrates both these approaches.

Debugging Nonreproducible Bugs

Fixing bugs that are not reproducible is significantly more difficult than fixing reproducible bugs. You often have very little information and must employ a lot of guesswork. Nevertheless, a few strategies can aid you.

1. Try to turn a nonreproducible bug into a reproducible bug. By using educated guesses, you can often determine approximately where the bug lies. It's worthwhile to spend some time trying to reproduce the bug. Once you have a reproducible bug you can figure out its root cause by using the techniques described earlier.
2. Analyze error logs. Easily done if you have instrumented your program with error log generation, as described earlier. You should sift through this information because any errors that were logged directly before the bug occurred are likely to have contributed to the bug itself. If you're lucky (or if you coded your program well), your program will have logged the exact reason for the bug at hand.
3. Obtain and analyze traces. Again, easily done if you have instrumented your program with tracing output, for example via a ring buffer as described earlier. At the time of the bug's

occurrence, you hopefully obtained a copy of the traces. These traces should lead you right to the location of the bug in your code.

4. Examine a *memory dump* file, if it exists. Some platforms generate memory dump files of applications that terminate abnormally. On Unix and Linux these memory dumps are called *core files*. Each platform provides tools for analyzing these memory dumps. They can, for example, be used to generate a stack trace of the application, or to view the contents of its memory before the application died.
5. Inspect the code. Unfortunately, this is often the only strategy to determine the cause of a nonreproducible bug. Surprisingly, it often works. When you examine code, even code that you wrote yourself, with the perspective of the bug that just occurred, you can often find mistakes that you overlooked previously. I don't recommend spending hours staring at your code, but tracing through the code path by hand can often lead you directly to the problem.
6. Use a memory-watching tool, such as one of those described in the "Debugging Memory Problems" section, which follows. Such tools often alert you to memory errors that don't always cause your program to misbehave, but could potentially be the cause of the bug at hand.
7. File or update a bug report. Even if you can't find the root cause of the bug right away, the report will be a useful record of your attempts if the problem is encountered again.

Once you have found the root cause of a nonreproducible bug, you should create a reproducible test case and move it to the "reproducible bugs" category. It is important to be able to reproduce a bug before you actually fix it. Otherwise, how will you test the fix? A common mistake when debugging nonreproducible bugs is to fix the wrong problem in the code. Because you can't reproduce the bug, you don't know if you've really fixed it, so don't be surprised when it shows up again a month later.

Debugging Regressions

If a feature contains a *regression* bug, it means that the feature used to work correctly, but stopped working due to the introduction of a bug.

A useful debugging technique for investigating regressions is to have a look at the change log of relevant files. If you know at what time the feature was still working, look at all the change logs since that time. You might notice something suspicious that could lead you to the root cause.

Another approach that can save you a lot of time debugging regressions is to use a binary search approach with older versions of the software to try and figure out when it started to go wrong. You can use binaries of older versions if you keep those, or revert the source code to an older revision with your source code server. Once you know when it started to go wrong, inspect the change logs to see what has changed at that time. This mechanism is only possible when you can reproduce the bug.

Debugging Memory Problems

Most catastrophic bugs, such as application death, are caused by memory errors. Many noncatastrophic bugs are triggered by memory errors as well. Some memory bugs are obvious.

If your program attempts to dereference a `nullptr` pointer, the default action is to terminate the program. However, nearly every platform gives you the capability of responding to catastrophic errors and taking remedial action. The amount of effort you devote to this depends on the importance of this kind of recovery to your end users. For example, a text editor really needs to make a best-attempt to save the modified buffers (possibly under a “recovered” name), while for other programs, users can find the default behavior acceptable, even if it is unpleasant.

Some memory bugs are more insidious. If you write past the end of an array in C++, your program will probably not crash directly at that point. However, if that array was on the stack, you may have written into a different variable or array, changing values that won’t show up until later in the program. Alternatively, if the array was on the heap, you could cause memory corruption in the heap, which will cause errors later when you attempt to allocate or free more memory dynamically.

Chapter 22 introduces some of the common memory errors from the perspective of what to avoid when you’re coding. This section discusses memory errors from the perspective of identifying problems in code that exhibits bugs. You should be familiar with the discussion in Chapter 22 before continuing with this section.

WARNING *Most of the following memory problems can be avoided by using smart pointers instead of dumb pointers.*

Categories of Memory Errors

In order to debug memory problems you should be familiar with the types of errors that can occur. This section describes the major categories of memory errors. Each memory error includes a small code example demonstrating the error and a list of possible *symptoms* that you might observe. Note that a symptom is not the same thing as a bug itself: A symptom is an observable behavior caused by a bug.

Memory-Freeing Errors

The following table summarizes five major errors involving freeing memory.

ERROR TYPE	SYMPTOMS	EXAMPLE
Memory leak	<p>Process memory usage grows over time.</p> <p>Process runs slower over time.</p> <p>Eventually, operations and system calls fail because of lack of memory.</p>	<pre>void memoryLeak() { int* p = new int[1000]; return; // Bug! Not freeing p. }</pre>

continues

(continued)

ERROR TYPE	SYMPTOMS	EXAMPLE
Using mismatched allocation and free operations	<p>Does not usually cause a crash immediately.</p> <p>Can cause memory corruption on some platforms, which might show up as a crash later in the program.</p> <p>Certain mismatches can also cause memory leaks.</p>	<pre>void mismatchedFree() { int* p1 = (int*)malloc(sizeof(int)); int* p2 = new int; int* p3 = new int[1000]; delete p1; // BUG! Should use free delete[] p2; // BUG! Should use delete free(p3); // BUG! Should use delete[] }</pre>
Freeing memory more than once	Can cause a crash if the memory at that location has been handed out in another allocation between the two calls to delete.	<pre>void doubleFree() { int* p1 = new int[1000]; delete[] p1; int* p2 = new int[1000]; delete[] p1; // BUG! freeing p1 twice } // BUG! Leaking memory of p2</pre>
Freeing unallocated memory	Will usually cause a crash.	<pre>void freeUnallocated() { int* p = reinterpret_cast<int*>(10000); delete p; // BUG! p not a valid pointer. }</pre>
Freeing stack memory	Technically a special case of freeing unallocated memory. Will usually cause a crash.	<pre>void freeStack() { int x; int* p = &x; delete p; // BUG! Freeing stack memory }</pre>

The crashes mentioned in this table can have different manifestations depending on your platform, such as segmentation faults, bus errors, or access violations.

As you can see, some of the errors do not cause immediate program termination. These bugs are more subtle, leading to problems later in the run of the program.

Memory-Access Errors

The second category of memory errors involves the actual reading and writing of memory.

ERROR TYPE	SYMPTOMS	EXAMPLE
Accessing invalid memory	Almost always causes program to crash immediately.	<pre>void accessInvalid() { int* p = reinterpret_cast<int*>(10000); *p = 5; // BUG! p is not a valid pointer. }</pre>
Accessing freed memory	Does not usually cause a program crash. If the memory has been handed out in another allocation, can cause "strange" values to appear unexpectedly.	<pre>void accessFreed() { int* p1 = new int; delete p1; int* p2 = new int; *p1 = 5; // BUG! The memory pointed to // by p1 has been freed. }</pre>
Accessing memory in a different allocation	Does not cause a crash. Can cause "strange" and potentially dangerous values to appear unexpectedly.	<pre>void accessElsewhere() { int x, y[10], z; x = 0; z = 0; for (int i = 0; i <= 10; i++) { y[i] = 5; // BUG for i==10! element 10 // is past end of array. } }</pre>
Reading uninitialized memory	Does not cause a crash unless you use the uninitialized value as a pointer and dereference it (as in the example). Even then, it will not always cause a crash.	<pre>void readUninitialized() { int* p; cout << *p; // BUG! p is uninitialized }</pre>

Memory-access errors don't always cause a crash. They can instead lead to subtle errors, in which the program does not terminate but instead produces erroneous results. Erroneous results can lead to serious consequences; for example, when external devices (such as robotic arms, X-ray machines, radiation treatments, life support systems, etc.) are being controlled by the computer.

Note that the discussed symptoms for memory-freeing errors and memory-access errors are the default symptoms for release builds of your program. Debug builds will most likely behave differently, and when run inside a debugger, the debugger might break into the code when an error occurs.

Tips for Debugging Memory Errors

Memory-related bugs often show up in slightly different places in the code each time you run the program. This is usually the case with heap memory corruption. Heap memory corruption is like a time bomb, ready to explode at some attempt to allocate, free, or use memory on the heap. So, when you see a bug that is reproducible, but shows up in slightly different places, suspect memory corruption.

If you suspect a memory bug, your best option is to use a memory-checking tool for C++. Debuggers often provide options to run the program while checking for memory errors. Additionally, there are some excellent third-party tools such as *purify* from Rational Software (now owned by IBM) or *valgrind* for Linux (discussed in Chapter 22). Microsoft provides a free download called *Application Verifier*, which can be used in a Windows environment. It is a run-time verification tool to help you find subtle programming errors like the previously discussed memory errors. These debuggers and tools work by interposing their own memory-allocation and -freeing routines in order to check for any misuse of dynamic memory, such as freeing unallocated memory, dereferencing unallocated memory, or writing off the end of an array.

If you don't have a memory-checking tool at your disposal, and the normal strategies for debugging are not helping, you may need to resort to code inspection. First, narrow down the part of the code containing the bug. Then, as a general rule, look at all naked pointers. Provided that you work on moderate to good quality code, most pointers should already be wrapped in smart pointers. If you do encounter naked pointers, take a closer look at how they are used, because they might be the cause of the error. Here are some more items to look for in your code.

Object and Class-related Errors

- Verify that your classes with dynamically allocated memory have destructors that free exactly the memory that's allocated in the object: no more, and no less.
- Ensure that your classes handle copying and assignment correctly with copy constructors and assignment operators, as described in Chapter 8. Make sure move constructors and move assignment operators properly set pointers in the source object to `nullptr` so that their destructors don't try to free that memory.
- Check for suspicious casts. If you are casting a pointer to an object from one type to another, make sure that it's valid. When possible, use `dynamic_cast`.

General Memory Errors

- Make sure that every call to `new` is matched with exactly one call to `delete`. Similarly, every call to `malloc`, `alloc`, or `calloc` should be matched with one call to `free`. And every call to

`new[]` should be matched with one call to `delete[]`. To avoid freeing memory multiple times or using freed memory, it's recommended to set your pointer to `nullptr` after freeing its memory.

- Check for buffer overruns. Anytime you iterate over an array or write into or read from a C-style string, verify that you are not accessing memory past the end of the array. These problems can often be avoided by using STL containers and strings.
- Check for dereferencing of invalid pointers.
- When declaring a pointer on the stack, make sure you always initialize it as part of its declaration; for example: `T* p = nullptr;` or `T* p = new T;` but never: `T* p;`
- Similarly, make sure your classes always initialize pointer data members in their constructors, by either allocating memory in the constructor or setting the pointers to `nullptr`.

Debugging Multithreaded Programs

C++ includes a threading library that provides mechanisms for threading and synchronization between threads. This threading library is discussed in Chapter 23. Multithreaded C++ programs are common, so it is important to think about the special issues involved in debugging a multithreaded program. Bugs in multithreaded programs are often caused by variations in timings in the operating system scheduling, and can be difficult to reproduce. Thus, debugging multithreaded programs takes a special set of techniques:

1. **Use a debugger:** A debugger makes it relatively easy to diagnose certain multithreaded problems; for example, deadlocks. When the deadlock appears, break into the debugger and inspect the different threads. You will be able to see which threads are blocked and on which line in the code they are blocked. Combining this with trace logs that show you how you came into the deadlock situation should be enough to fix deadlocks.
2. **Use log-based debugging:** When debugging multithreaded programs, log-based debugging can sometimes be more effective than using a debugger to debug certain problems. You can add log statements to your program before and after critical sections, and before acquiring and after releasing locks. Log-based debugging is extremely useful to investigate race conditions. However, the act of adding log statements slightly changes run-time timings, which might hide the bug.
3. **Insert forced sleeps and context switches:** If you are having trouble reproducing the problem consistently, or have a hunch about the root cause but want to verify it, you can force certain thread-scheduling behavior by making your threads sleep for specified amounts of time. The `<thread>` header defines `sleep_until()` and `sleep_for()` in the `std::this_thread` namespace, which you can use to sleep. Sleeping for several seconds right before releasing a lock, immediately before signaling a condition variable, or directly before accessing shared data can reveal race conditions that would otherwise go undetected. If this debugging technique reveals the root cause, it must be fixed, so that it works correctly after removing these forced sleeps and context switches. Leaving these forced sleeps and context switches in place to “fix” the problem is wrong.

4. **Perform code review:** Reviewing your thread synchronization code often helps in fixing race conditions. Try to proof over and over that what happened is not possible, until you see how it is. It doesn't hurt to write down these "proofs" in code comments. Also, ask a coworker to do pair debugging; he or she might see something you are overlooking.

Debugging Example: Article Citations

This section presents a buggy program and shows you the steps to take in order to debug it and fix the problem.

Suppose that you're part of a team writing a web page that allows users to search for the research articles that cite a particular paper. This type of service is useful for authors who are trying to find work similar to their own. Once they find one paper representing a related work, they can look for every paper that cites that one to find other related work.

In this project, you are responsible for the code that reads the raw citations data from text files. For simplicity, assume that the citation info for each paper is found in its own file. Furthermore, assume that the first line of each file contains the author, title, and publication info for the paper; the second line is always empty; and all subsequent lines contain the citations from the article (one on each line). Here is an example file for one of the most important papers in computer science:

Alan Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936-37), 230-265.

Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.

Alonzo Church. "An unsolvable problem of elementary number theory", American J. of Math., 58 (1936), 345-363.

Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic Logic, 1 (1936), 40-41.

E.W. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87-88.

Buggy Implementation of an ArticleCitations Class

You decide to structure your program by writing an `ArticleCitations` class that reads the file and stores the information. This class stores the article info from the first line in one string, and the citations' info in an array of strings. Please note that this design decision is a bad one. You should opt for one of the STL containers to store the citations. There are other obvious issues with this implementation, such as using `int` instead of `size_t`. However, for the purpose of illustrating buggy applications, it's perfect. The class definition looks like this:

```
class ArticleCitations
{
public:
    ArticleCitations(const std::string& fileName);
    virtual ~ArticleCitations();
    ArticleCitations(const ArticleCitations& src);
    ArticleCitations& operator=(const ArticleCitations& rhs);
    const std::string& getArticle() const { return mArticle; }
    int getNumCitations() const { return mNumCitations; }
```

```

        const std::string& getCitation(int i) const { return mCitations[i]; }
private:
    void readFile(const std::string& fileName);
    void copy(const ArticleCitations& src);
    std::string mArticle;
    std::string* mCitations;
    int mNumCitations;
};

```

The implementation follows. This program is buggy! Don't use it verbatim or as a model.

```

ArticleCitations::ArticleCitations(const string& fileName)
    : mCitations(nullptr), mNumCitations(0)
{
    // All we have to do is read the file.
    readFile(fileName);
}
ArticleCitations::ArticleCitations(const ArticleCitations& src)
{
    copy(src);
}
ArticleCitations& ArticleCitations::operator=(const ArticleCitations& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return *this;
    }
    // Free the old memory.
    delete [] mCitations;
    // Copy the data
    copy(rhs);
    return *this;
}
void ArticleCitations::copy(const ArticleCitations& src)
{
    // Copy the article name, author, etc.
    mArticle = src.mArticle;
    // Copy the number of citations
    mNumCitations = src.mNumCitations;
    // Allocate an array of the correct size
    mCitations = new string[mNumCitations];
    // Copy each element of the array
    for (int i = 0; i < mNumCitations; i++) {
        mCitations[i] = src.mCitations[i];
    }
}
ArticleCitations::~ArticleCitations()
{
    delete [] mCitations;
}
void ArticleCitations::readFile(const string& fileName)
{
    // Open the file and check for failure.
    ifstream istr(fileName.c_str());
    if (istr.fail()) {

```

```

        throw invalid_argument("Unable to open file");
    }
    // Read the article author, title, etc. line.
    getline(istr, mArticle);
    // Skip the white space before the citations start.
    istr >> ws;
    int count = 0;
    // Save the current position so we can return to it.
    ios_base::streampos citationsStart = istr.tellg();
    // First count the number of citations.
    while (!istr.eof()) {
        // Skip white space before the next entry.
        istr >> ws;
        string temp;
        getline(istr, temp);
        if (!temp.empty()) {
            count++;
        }
    }
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            if (!temp.empty()) {
                mCitations[count] = temp;
            }
        }
    } else {
        mNumCitations = -1;
    }
}
}

```

Testing the ArticleCitations class

You decide to test your ArticleCitations class. The following program asks the user for a filename, constructs an ArticleCitations class with that filename, and passes the object by value to the processCitations() function, which prints out the info using the public accessor methods on the object:

```

void processCitations(ArticleCitations cit)
{
    cout << cit.getArticle() << endl;
    int num = cit.getNumCitations();
    for (int i = 0; i < num; i++) {
        cout << cit.getCitation(i) << endl;
    }
}
int main()

```

```

{
    string fileName;
    while (true) {
        cout << "Enter a file name (\\"STOP\\" to stop): ";
        cin >> fileName;
        if (fileName == "STOP") {
            break;
        }
        // Test constructor
        ArticleCitations cit(fileName);
        processCitations(cit);
    }
    return 0;
}

```

Message-Based Debugging

You decide to test the program on the Alan Turing example (stored in a file called `paper1.txt`). Here is the output:

```

Enter a file name ("STOP" to stop): paper1.txt
Alan Turing, "On Computable Numbers, with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

That doesn't look right. There are supposed to be four citations printed instead of four blank lines.

For this bug, you decide to try log-based debugging, and because this is a console example, you decide to just print messages to `cout`. In this case, it makes sense to start by looking at the function that reads the citations from the file. If that doesn't work right, then obviously the object won't have the citations. You can modify `readFile()` as follows:

```

void ArticleCitations::readFile(const string& fileName)
{
    // Code omitted for brevity
    // First count the number of citations.
    cout << "readFile(): counting number of citations" << endl;
    while (!istr.eof()) {
        // Skip white space before the next entry.
        istr >> ws;
        string temp;
        getline(istr, temp);
        if (!temp.empty()) {
            cout << "Citation " << count << ":" << temp << endl;
            count++;
        }
    }
    cout << "Found " << count << " citations" << endl;
    cout << "readFile(): reading citations" << endl;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
    }
}

```

```

mNumCitations = count;
// Seek back to the start of the citations.
istr.seekg(citationsStart);
// Read each citation and store it in the new array.
for (count = 0; count < mNumCitations; count++) {
    string temp;
    getline(istr, temp);
    if (!temp.empty()) {
        cout << temp << endl;
        mCitations[count] = temp;
    }
}
} else {
    mNumCitations = -1;
}
cout << "readFile(): finished" << endl;
}
}

```

Running the same test with this program gives the following output:

```

Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
Citation 0: Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und
verwandter Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.
Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory",
American J. of Math., 58 (1936), 345-363.
Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic
Logic, 1 (1936), 40-41.
Citation 3: E.W. Hobson, "Theory of functions of a real variable (2nd ed., 1921)",
87-88.
Found 4 citations
readFile(): reading citations
readFile(): finished
Alan Turing, "On Computable Numbers, with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

As you can see from the output, the first time the program reads the citations from the file, in order to count them, they are read correctly. However, the second time, they are not read correctly; Nothing is printed between “readFile(): reading citations” and “readFile(): finished”. Why not? One way to delve deeper into this issue is to add some debugging code to check the state of the file stream after each attempt to read a citation:

```

void printStreamState(const istream& istr)
{
    if (istr.good()) {
        cout << "stream state is good" << endl;
    }
    if (istr.bad()) {
        cout << "stream state is bad" << endl;
    }
}

```

```

        if (istr.fail()) {
            cout << "stream state is fail" << endl;
        }
        if (istr.eof()) {
            cout << "stream state is eof" << endl;
        }
    }
void ArticleCitations::readFile(const string& fileName)
{
    // Code omitted for brevity
    // First count the number of citations.
    cout << "readFile(): counting number of citations" << endl;
    while (!istr.eof()) {
        // Skip white space before the next entry.
        istr >> ws;
        string temp;
        getline(istr, temp);
        printStreamState(istr);
        if (!temp.empty()) {
            cout << "Citation " << count << ": " << temp << endl;
            count++;
        }
    }
    cout << "Found " << count << " citations" << endl;
    cout << "readFile(): reading citations" << endl;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            printStreamState(istr);
            if (!temp.empty()) {
                cout << temp << endl;
                mCitations[count] = temp;
            }
        }
    } else {
        mNumCitations = -1;
    }
    cout << "readFile(): finished" << endl;
}

```

When you run your program this time, you find some interesting information:

```

Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
stream state is good
Citation 0: Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und
verwandter Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.
stream state is good

```

```

Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory",
American J. of Math., 58 (1936), 345-363.
stream state is good
Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic
Logic, 1 (1936), 40-41.
stream state is good
Citation 3: E.W. Hobson, "Theory of functions of a real variable (2nd ed., 1921)",
87-88.
stream state is fail
stream state is eof
Found 4 citations
readFile(): reading citations
stream state is fail
readFile(): finished
Alan Turing, "On Computable Numbers, with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

It looks like the stream state is good until after the final citation is read for the first time. Then, the stream state is fail and eof, because the end-of-file has been reached and `istr >> ws` still tries to read some white space. That is expected. What is not expected is that the stream state remains fail after all attempts to read the citations a second time. That doesn't appear to make sense at first: the code uses `seekg()` to seek back to the beginning of the citations before reading them a second time.

However, Chapter 12 explains that streams maintain their error states until you clear them explicitly; `seekg()` doesn't clear the fail state automatically. When in an error state, streams fail to read data correctly, which explains why the stream state is fail also after trying to read the citations a second time. A closer look at your method reveals that it fails to call `clear()` on the `istream` after reaching the end of the file. If you modify the method by adding a call to `clear()`, it will read the citations properly.

Here is the corrected `readFile()` method without the debugging `cout` statements:

```

void ArticleCitations::readFile(const string& fileName)
{
    // Code omitted for brevity
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Clear the stream state.
        istr.clear();
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            if (!temp.empty()) {

```

Running the same test again on `paper1.txt` now shows you the correct four citations.

Using the GDB Debugger on Linux

Now that your ArticleCitations class seems to work well on one citations file, you decide to blaze ahead and test some special cases, starting with a file with no citations. The file looks like this, and is stored in a file named `paper2.txt`:

Author with no citations

When you try to run your program on this file, depending on the version of your Linux and your compiler, you might get a crash that looks something like the following:

```
Enter a file name ("STOP" to stop): paper2.txt
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

The message “core dumped” means that the program crashed. This time you decide to give the debugger a shot. The Gnu DeBugger (gdb) is widely available on Unix and Linux platforms. First, you must compile your program with debugging info (-g with g++). Then you can launch the program under gdb. Here’s an example session using the debugger to find the root cause of this problem. This example assumes your compiled executable is called `buggyprogram`. Text that you have to type is shown in bold.

```
> gdb buggyprogram
[ Start-up messages omitted for brevity ]
Reading symbols from /home/marc/c++/gdb/buggyprogram...done.
(gdb) run
Starting program: buggyprogram
Enter a file name ("STOP" to stop): paper2.txt
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Program received signal SIGABRT, Aborted.
0x00007ffff7535c39 in raise () from /lib64/libc.so.6
(gdb)
```

When the program crashes, the debugger breaks the execution, and allows you to poke around in the state of the program at that time. The `backtrace` or `bt` command shows the current stack trace. The last operation is at the top, with frame number zero, #0:

```
(gdb) bt
#0  0x00007ffff7535c39 in raise () from /lib64/libc.so.6
#1  0x00007ffff7537348 in abort () from /lib64/libc.so.6
```

```

#2 0x00007ffff7b35f85 in __gnu_cxx::__verbose_terminate_handler() () from /lib64/
libstdc++.so.6
#3 0x00007ffff7b33ee6 in ?? () from /lib64/libstdc++.so.6
#4 0x00007ffff7b33f13 in std::terminate() () from /lib64/libstdc++.so.6
#5 0x00007ffff7b3413f in __cxa_throw () from /lib64/libstdc++.so.6
#6 0x00007ffff7b346cd in operator new(unsigned long) () from /lib64/libstdc++.so.6
#7 0x00007ffff7b34769 in operator new[](unsigned long) () from /lib64/libstdc++.
so.6
#8 0x00000000004016ea in ArticleCitations::copy (this=0x7fffffff090, src=...) at
ArticleCitations.cpp:40
#9 0x00000000004015b5 in ArticleCitations::ArticleCitations (this=0x7fffffff090,
src=...)
    at ArticleCitations.cpp:16
#10 0x0000000000401d0c in main () at ArticleCitationsTest.cpp:20

```

When you get a stack trace like this, you should try to find the first stack frame from the top that is in your own code. In this example, this is stack frame #8. From this frame you can see that there seems to be some sort of problem in the `copy()` method of `ArticleCitations`. This method is invoked because `main()` calls `processCitations()` and passes the argument by value, which triggers a call to the `copy` constructor, which calls `copy()`. Of course, in production code you should pass a `const` reference, but pass-by-value is used for this example of a buggy program. You can tell the debugger to switch to stack frame #8 with the `frame` command, which requires the index of the frame to jump to:

```

(gdb) frame 8
#8 0x00000000004016ea in ArticleCitations::copy (this=0x7fffffff090, src=...) at
ArticleCitations.cpp:40
40    mCitations = new string[mNumCitations];

```

This output shows that the following line caused a problem:

```
mCitations = new string[mNumCitations];
```

Now, use the `list` command to show the code in the current stack frame around the offending line:

```

(gdb) list
35    // Copy the article name, author, etc.
36    mArticle = src.mArticle;
37    // Copy the number of citations
38    mNumCitations = src.mNumCitations;
39    // Allocate an array of the correct size
40    mCitations = new string[mNumCitations];
41    // Copy each element of the array
42    for (int i = 0; i < mNumCitations; i++) {
43        mCitations[i] = src.mCitations[i];
44    }

```

In gdb, you can print values available in the current scope with the `print` command. In order to find the root cause of the problem, you can try printing some of the variables. The error happens inside the `copy()` method, so checking the value of the `src` parameter is a good start:

```
(gdb) print src
$1 = (const ArticleCitations &) @0x7fffffff060: {
    _vptr.ArticleCitations = 0x401fb0 <vtable for ArticleCitations+16>,
    mArticle = "Author with no citations", mCitations = 0x7fffffff080, mNumCitations
    = -1}
```

Ah-ha! Here's the problem. This article isn't supposed to have any citations. Why is `mNumCitations` set to the strange value `-1`? Take another look at the code in `readFile()` for the case that there are no citations. In that case, it looks like `mNumCitations` is erroneously set to `-1`. The fix is easy, you need to initialize `mNumCitations` to `0`, instead of setting it to `-1` when there are no citations. Another problem, `readFile()` can be called multiple times on the same `ArticleCitations` object, so you also need to free a previously allocated `mCitations` array. Here is the fixed code:

```
void ArticleCitations::readFile(const string& fileName)
{
    // Code omitted for brevity
    delete [] mCitations; // Free previously allocated citations.
    mCitations = nullptr;
    mNumCitations = 0;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Clear the previous eof.
        istr.clear();
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        // Code omitted for brevity
    }
}
```

As this example shows, bugs don't always show up right away. It often takes a debugger and some persistence to figure them out.

NOTE *If you attempt to replicate this debugging session on a different platform, you may find that, due to the vagaries of memory errors, the program crashes in a different place than this example shows.*

Using the Visual C++ 2013 Debugger

This section explains the same debugging procedure as described in the previous section, but uses the Microsoft Visual C++ 2013 debugger instead of `gdb`.

First, you need to create a project. Start VC++ and click on File \Rightarrow New \Rightarrow Project. In the project template tree on the left, select Visual C++ \Rightarrow Win32. Then select the Win32 Console

Application template in the list in the middle of the window. At the bottom you can give a name for the project and a location where to save it. Specify ArticleCitations as the name, choose a folder where to save the project, and click OK. A wizard opens. Click Next, select Console application and Empty Project, and click Finish.

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, go to View \Rightarrow Solution Explorer. Right-click the ArticleCitations project in the Solution Explorer and click Add \Rightarrow Existing Item. Add all the files from the ArticleCitations\05_VisualStudio folder in the downloadable code archive to the project. Your Solution Explorer should look similar to Figure 26-1.

Now you can compile the program; click Build \Rightarrow Build Solution. Copy the paper1.txt and paper2.txt test files to your ArticleCitations project folder, which is the folder containing the ArticleCitations.vcxproj file.

Run the application with Debug \Rightarrow Start Debugging, and test the program by first specifying the paper1.txt file. It should properly read the file and output the result to the console. Then, type paper2.txt. A Microsoft Visual C++ Runtime Library message will be displayed with three buttons: Abort, Retry, and Ignore. Click Retry, which causes the VC++ debugger to break the execution. You will get a message saying “ArticleCitations.exe has triggered a breakpoint.” in which you need to click Break.

At this point, you should inspect the call stack, Debug \Rightarrow Windows \Rightarrow Call Stack. In this call stack, you need to find the first line that contains code that you wrote. This is shown in Figure 26-2.

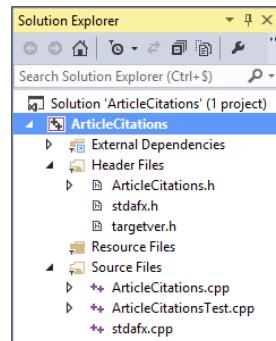


FIGURE 26-1

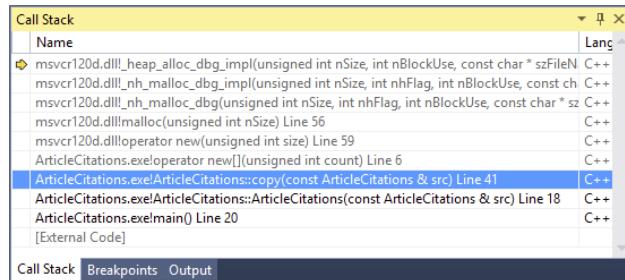


FIGURE 26-2

Just as with gdb, you see that the problem is in `copy()`. You can double-click that line in the call stack window to jump to the right place in the code. If you only see disassembly code, right click anywhere on the disassembly and select Go To Source Code. Then click Debug \Rightarrow Windows \Rightarrow Autos to inspect variables. In the list of variables you can find `src`. Click the plus sign to expand the data members of the `src` variable. Figure 26-3 shows how it looks.

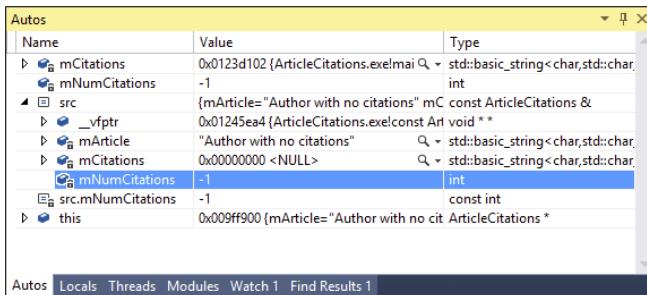


FIGURE 26-3

From this window, you see that `mNumCitations` is `-1`. The reason and the fix are exactly the same as earlier.

Lessons from the ArticleCitations Example

You might be inclined to disregard this example as too small to be representative of real debugging. Although the buggy code is not lengthy, many classes that you write will not be much bigger, even in large projects. Imagine if you had failed to test this example thoroughly before integrating it with the rest of the project. If these bugs showed up later, you and other engineers would have to spend more time narrowing down the problem before you could debug it as shown here. Additionally, the techniques shown in this example apply to all debugging, large or small scale.

SUMMARY

The most important concept in this chapter was the Fundamental Law of Debugging: *avoid bugs when you're coding, but plan for bugs in your code*. The reality of programming is that bugs will appear. If you've prepared your program properly, with error logging, debug traces, assertions, and static assertions, then the actual debugging will be significantly easier.

In addition to these techniques, this chapter also presented specific approaches for debugging bugs. The most important rule when actually debugging is to reproduce the problem. Then, you can use a symbolic debugger or log-based debugging to track down the root cause. Memory errors present particular difficulties, and account for the majority of bugs in C++ code. This chapter described the various categories of memory bugs and their symptoms, and showed examples of debugging errors in a program.

Debugging techniques are a great way to end your journey through *Professional C++*. By thinking through your designs, experimenting with different approaches in object-oriented programming, selectively adding new techniques to your coding repertoire, and practicing debugging techniques, you'll be able to take your C++ skills to the professional level.

A

C++ Interviews

Reading this book will surely give your C++ career a kick-start, but employers will want you to prove yourself before they offer the big bucks. Interview methodologies vary from company to company, but many aspects of technical interviews are predictable. A thorough interviewer will want to test your basic coding skills, your debugging skills, your design and style skills, and your problem-solving skills. The set of questions you might be asked is quite large. In this appendix, you'll read about some of the different types of questions you may encounter and the best tactics for landing that high-paying C++ programming job you're after.

This appendix iterates through the chapters of the book, discussing the aspects of each chapter likely to come up in an interview situation. Each section also includes a discussion of the types of questions that could be designed to test those skills, and the best ways to deal with those questions.

CHAPTER 1: A CRASH COURSE IN C++ AND THE STL

A technical interview will often include some basic C++ questions to weed out the candidates who put C++ on their resume simply because they've heard of the language. These questions might be asked during a *phone screen*, when a developer or recruiter calls you before bringing you in for an in-person interview. They could also be asked via e-mail or in person. When answering these questions, remember that the interviewer is just trying to establish that you've actually learned and used C++. You generally don't need to get every detail right to earn high marks.

Things to Remember

- Use of functions
- Header file syntax, including the omission of ".h" for standard library headers
- Basic use of namespaces
- Language basics, such as loop syntax, including the range-based `for` loop, the conditional operator, and variables

- The difference between the stack and the heap
- Dynamically allocated arrays
- Use of `const`
- What references are
- The `auto` keyword
- Basic use of STL containers such as `std::vector`

Types of Questions

Basic C++ questions will often come in the form of a vocabulary test. The interviewer may ask you to define C++ terms, such as `const` or `static`. He or she may be looking for the textbook answer, but you can often score extra points by giving sample usage or extra detail. For example, in addition to saying that one of the uses of `const` is to specify that a reference argument cannot be changed, you can also say that a `const` reference is more efficient than a copy when passing an object into a function or method.

The other form that basic C++ competence questions can take is a short program that you write in front of the interviewer. An interviewer may give you a warm-up question, such as, “Write *Hello, World* in C++.” When you get a seemingly simple question like this, make sure that you score all the extra points you can by showing that you are namespace-savvy, you use streams instead of `printf()`, and you know which standard headers to include.

CHAPTERS 2 AND 18: STRINGS, LOCALIZATION, AND REGULAR EXPRESSIONS

Strings are very important, and are used in almost every kind of application. An interviewer will most likely ask at least one question related to string handling in C++.

Things to Remember

- The `std::string` class
- Differences between the C++ `std::string` class and C-style strings, including why C-style strings should be avoided
- Conversion of strings to numeric types such as integers and floating point numbers, and vice versa
- Raw string literals
- The importance of localization
- Ideas behind Unicode
- The concepts of locales and facets
- What regular expressions are

Types of Questions

An interviewer could ask you to explain how you can append two strings together. With this question he or she wants to find out whether you are thinking as a professional C++ programmer or as a C programmer. If you get such a question, you should explain the `std::string` class, and show how to use it to append two strings. It's also worth mentioning that the `string` class will handle all memory management for you automatically and contrast this to C-style strings.

Your interviewer may not ask specifically about localization, but you can show your worldwide interest by using `wchar_t` instead of `char` during the interview. If you do receive a question about your experience with localization, be sure to mention the importance of considering worldwide use from the beginning of the project.

You may also be asked about the general idea behind locales and facets. You probably will not have to explain the exact syntax, but you should explain that they allow you to format text and numbers according to the rules of a certain language/country.

You might get a question about Unicode, but most likely it will be a question to explain the ideas and the basic concepts behind Unicode instead of implementation details. So, make sure you understand the high-level concepts of Unicode and that you can explain their use in the context of localization. You should also know about the different options for encoding Unicode characters, such as UTF-8 and UTF-16, without specific details.

As seen in Chapter 18, regular expressions can have a daunting syntax. It is unlikely that an interviewer will ask you about little details of regular expressions. However, you should be able to explain the concept of regular expressions and what kind of string manipulations you can do with them.

CHAPTER 3: CODING WITH STYLE

Anybody who's coded in the professional world has had a co-worker who codes as if they learned C++ from the back of a cereal box. Nobody wants to work with someone who writes messy code, so interviewers sometimes attempt to determine a candidate's style skills.

Things to Remember

- Style matters, even during interview questions that aren't explicitly style related
- Well-written code doesn't need extensive comments
- Comments can be used to convey meta information
- The principle of decomposition
- The principle of refactoring
- Naming techniques

Types of Questions

Style questions can come in a few different forms. A friend of mine was once asked to write the code for a relatively complex algorithm on a whiteboard. As soon as he wrote the first variable name, the interviewer stopped him and told him he passed. The question wasn't about the algorithm; it was just a red herring to see how well he named his variables. More commonly, you may be asked to submit code that you've written, or to give your opinions on style.

You need to be careful when a potential employer asks you to submit code. You probably cannot legally submit code that you wrote for a previous employer. You also have to find a piece of code that shows off your skills without requiring too much background knowledge. For example, you wouldn't want to submit your master's thesis on high-speed image rendering to a company that is interviewing you for a database administration position.

If the company gives you a specific program to write, that's a perfect opportunity to show off what you've learned in this book. Even if the potential employer doesn't specify the program, you should consider writing a small program specifically to submit to the company. Instead of selecting some code you've already written, start from scratch to produce code that is relevant to the job and highlights good style.

Also, if you have documentation that you have written and that can be released, meaning it is not confidential, use it to show your skills to *communicate*; it will give you extra points. Websites you have built or maintained, articles you have submitted to places like CodeGuru, CodeProject, SourceForge, and so on, are very useful. It says you can not only *write code*, but also *communicate* to others how to effectively *use* that code. Of course, having a book title attached to your name is also a big plus.

CHAPTER 4: DESIGNING PROFESSIONAL C++ PROGRAMS

Your interviewer will want to make sure that in addition to knowing the C++ language, you are skilled at applying it. You may not be asked a design question explicitly, but good interviewers have a variety of techniques to sneak design into other questions, as you'll see.

A potential employer will also want to know that you're able to work with code that you didn't write yourself. If you've listed specific libraries on your resume, you should be prepared to answer questions on those. If you didn't list specific libraries, a general understanding of the importance of libraries will probably suffice.

Things to Remember

- Design is subjective — be prepared to defend design decisions you make during the interview
- Recall the details of a design you've done in the past prior to the interview in case you are asked for an example
- Be prepared to define *abstraction* and give an example

- Be prepared to sketch out a design visually, including class hierarchies
- Be prepared to tout the benefits of code reuse
- The concept of libraries
- The tradeoffs between building from scratch and reusing existing code
- The basics of big-O notation, or at least remember that $O(n \log n)$ is better than $O(n^2)$
- The functionality that is included in the C++ Standard Library
- The high-level definition of design patterns

Types of Questions

Design questions are hard for an interviewer to come up with — any program that you could design in an interview setting is probably too simple to demonstrate real-world design skills. Design questions may come in a more fuzzy form, such as, “Tell me the steps in designing a good program,” or “Explain the principle of abstraction.” They can also be less explicit. When discussing your previous job, the interviewer can say, “Can you explain the design of that project to me?”

If the interviewer is asking you about a specific library, he or she will probably focus on the high-level aspects of the library as opposed to technical specifics. For example, you can be asked to explain what the strengths and weaknesses of the STL are from a library design point of view. The best candidates talk about the STL’s breadth and standardization as strengths, and its steep learning curve as the major drawback.

You may also be asked a design question that initially doesn’t sound as if it’s related to libraries. For example, the interviewer could ask how you would go about creating an application that downloads MP3 music from the web and plays it on the local computer. This question isn’t explicitly related to libraries, but that’s what it’s getting at; the question is really asking about process.

You should begin by talking about how you would gather requirements and do initial prototypes. Because the question mentions two specific technologies, the interviewer would like to know how you would deal with them. This is where libraries come into play. If you tell the interviewer that you would write your own web classes and MP3 playing code, you won’t fail the test, but you will be challenged to justify the time and expense of reinventing these tools.

A better answer would be to say that you would survey existing libraries that perform web and MP3 functionality to see if one exists that suits the project. You might want to name some technologies that you would start with, such as libcurl for web retrieval in Linux or the Windows Media library for music playback in Windows.

Mentioning some websites with free libraries, and some ideas of what those websites provide, might also get you extra points. For example, www.codeguru.com and www.codeproject.com for Windows libraries; www.boost.org for platform independent C++ libraries; www.sourceforge.net for libraries for different platforms; and so on. Giving examples of some of the licenses that are available for open-source software, such as the GPL license, Boost license, Creative Commons license, CodeGuru license, OpenBSD license, and so on, might score you extra credit.

CHAPTER 5: DESIGNING WITH OBJECTS

Object-oriented design questions are used to weed out C programmers who merely know what a reference is, from C++ programmers who actually use the object-oriented features of the language. Interviewers don't take anything for granted; even if you've been using object-oriented languages for years, they may still want to see evidence that you understand the methodology.

Things to Remember

- The differences between the procedural and object-oriented paradigms
- The differences between a class and an object
- Expressing classes in terms of components, properties, and behaviors
- Is-a and has-a relationships
- The tradeoffs involved in multiple inheritance

Types of Questions

There are typically two ways to ask object-oriented design questions. You can be asked to define an object-oriented concept, or you can be asked to sketch out an object-oriented hierarchy. The former is pretty straightforward. Remember that examples might earn you extra credit.

If you're asked to sketch out an OO hierarchy, the interviewer will usually provide a simple application, such as a card game, for which you should design a class hierarchy. Interviewers often ask design questions about games because those are applications with which most people are already familiar. They also help lighten the mood a bit when compared to questions about things like database implementations. The hierarchy you generate will, of course, vary based on the game or application they are asking you to design. Here are some points to consider:

- The interviewer wants to see your thought process. Think aloud, brainstorm, engage the interviewer in a discussion, and don't be afraid to erase and go in a different direction.
- The interviewer may assume that you are familiar with the application. If you've never heard of blackjack and you get a question about it, ask the interviewer to clarify or change the question.
- Unless the interviewer gives you a specific format to use when describing the hierarchy, it's recommended that your class diagrams take the form of inheritance trees with rough lists of methods and data members for each class.
- You may have to defend your design or revise it to take added requirements into consideration. Try to gauge whether the interviewer sees actual flaws in your design, or whether she just wants to put you on the defensive to see your skills of persuasion.

CHAPTER 6: DESIGNING FOR REUSE

Interviewers rarely ask questions about designing reusable code. This omission is unfortunate because having programmers on staff who can write only single-purpose code can be detrimental to a programming organization. Occasionally, you'll find a company that is savvy on code reuse and asks

about it in their interviews. Such a question is an indication that it might be a good company to work for.

Things to Remember

- The principle of abstraction
- The creation of subsystems and class hierarchies
- The general rules for good interface design, which are interfaces with only `public` methods and no implementation details
- When to use templates and when to use inheritance

Types of Questions

Questions about reuse will almost certainly be about previous projects on which you have worked. For example, if you worked at a company that produced both consumer and professional video-editing applications, the interviewer may ask how code was shared between the two applications. Even if you aren't explicitly asked about code reuse, you might be able to sneak it in. When you're describing some of your past work, tell the interviewer if the modules you wrote were used in other projects. Even when answering apparently straight coding questions, make sure to consider and mention the interfaces involved.

CHAPTERS 7 AND 8: CLASSES AND OBJECTS

There are no bounds to the types of questions you can be asked about classes and objects. Some interviewers are syntax-fixated and might throw some complicated code at you. Others are less concerned with the implementation and more interested in your design skills.

Things to Remember

- Basic class definition syntax
- Access specifiers for methods and data members
- The use of the `this` pointer
- How name resolution works, which resolves a name first by local scope, then class scope (implying `this->`), and then global scope
- Object creation and destruction, both on the stack and the heap
- Cases when the compiler generates a constructor for you
- Constructor initializers
- Copy constructor and assignment operator
- Delegating constructors
- The `mutable` keyword
- Method overloading and default parameters

- Friend classes and methods
- Managing dynamically allocated memory in objects
- `static` methods and data members
- Inline methods and the fact that the `inline` keyword is just a hint for the compiler, which can ignore the hint
- The key idea of separating interface and implementation classes, which says that interfaces should only contain `public` methods, and should be as stable as possible; they should not contain any data members or `private/protected` methods; thus, interfaces can remain stable while implementations are free to change under them.
- Initializer lists
- In-class member initializers
- Explicitly defaulted and deleted special member functions

Types of Questions

Questions such as, “What does the keyword `mutable` mean?” make great phone screening questions. A recruiter may have a list of C++ terms and will move candidates to the next stage of the process based on the number that they get right. You may not know all of the terms thrown at you, but keep in mind that other candidates are facing the same questions and it’s one of the few metrics available to a recruiter.

The find-the-bug style of questions is popular among interviewers and course instructors alike. You will be presented with some nonsense code and asked to point out its flaws. Interviewers struggle to find quantitative ways to analyze candidates, and this is one of the few ways to do it. In general, your approach should be to read each line of code and voice your concerns, brainstorming aloud. The types of bugs can fall into these categories:

- **Syntax errors:** These are rare — interviewers know you can find compile-time bugs with a compiler.
- **Memory problems:** These include problems such as leaks and double deletion.
- **“You wouldn’t do that” problems:** This category includes things that are technically correct but are not recommended. For example, don’t use C-style character arrays, use `std::string` instead.
- **Style errors:** Even if the interviewer doesn’t count it as a bug, point out poor comments or variable names.

Here’s a find-the-bug problem that demonstrates each of these areas:

```
class Buggy
{
    Buggy(int param);
    ~Buggy();
    double fjord(double inVal);
    int fjord(double inVal);
protected:
    void turtle(int i = 7, int j);
```

```

        int param;
        double* mGraphicDimension;
    };
Buggy::Buggy(int param)
{
    param = param;
    mGraphicDimension = new double;
}
Buggy::~Buggy()
{
}
double Buggy::fjord(double inVal)
{
    return inVal * param;
}
int Buggy::fjord(double inVal)
{
    return (int)fjord(inVal);
}
void Buggy::turtle(int i, int j)
{
    cout << "i is " << i << ", j is " << j << endl;
}

```

Take a careful look at the code, and then consult the following corrected version for the answers:

```

#include <iostream>           // Streams are used in the implementation.
class Buggy
{
    public:                  // These should most likely be public.
        Buggy(int inParam); // Parameter naming.
        virtual ~Buggy(); // Recommended to make destructors virtual.
        Buggy(const Buggy& src); // Provide copy ctor and operator=
        Buggy& operator=(const Buggy& rhs); // when the class has dynamically
                                              // allocated memory.
        Buggy(Buggy&& src) noexcept; // Provide move ctor and operator=
        Buggy& operator=(Buggy&& rhs) noexcept; // when the class has dynamically
                                              // allocated memory to increase
                                              // performance. (See Chapter 10).
        double fjord(double inVal); // int version won't compile. Overloaded
                                     // methods cannot differ only in return type.
    private:                 // Use private by default.
        void turtle(int i, int j); // Only last arguments can have defaults.
        int mParam;             // Data member naming.
        double* mGraphicDimension;
    };
Buggy::Buggy(int inParam) : mParam(inParam) // Prefer using the ctor initializer
{
    mGraphicDimension = new double;
}
Buggy::~Buggy()
{
    delete mGraphicDimension; // Avoid memory leak.
    mGraphicDimension = nullptr;
}

```

```
Buggy::Buggy(const Buggy& src)
{
    mParam = src.mParam;
    mGraphicDimension = new double;
    *mGraphicDimension = *(src.mGraphicDimension);
}
Buggy& Buggy::operator=(const Buggy& rhs)
{
    if (this == &rhs) {
        return *this;
    }
    mParam = rhs.mParam;
    delete mGraphicDimension;
    mGraphicDimension = new double;
    *mGraphicDimension = *(rhs.mGraphicDimension);
    return *this;
}
Buggy::Buggy(Buggy&& src) noexcept
{
    mParam = src.mParam;
    mGraphicDimension = src.mGraphicDimension;
    src.mGraphicDimension = nullptr;
}
Buggy& Buggy::operator=(Buggy&& rhs) noexcept
{
    if (this == &rhs) {
        return *this;
    }
    mParam = rhs.mParam;
    mGraphicDimension = rhs.mGraphicDimension;
    rhs.mGraphicDimension = nullptr;
    return *this;
}
double Buggy::fjord(double inVal)
{
    return inVal * mParam;    // Changed data member name.
}
void Buggy::turtle(int i, int j)
{
    std::cout << "i is " << i << ", j is " << j << std::endl; // Namespaces.
}
```

For this example, you *must* also mention that it's better to avoid dynamic memory, or, if you can't avoid it, to use a smart pointer, such as a `unique_ptr`, instead of the dumb `mGraphicDimension` pointer. Include an explanation of why a smart pointer is required, and explain the impact on the implementation of the `Buggy` class.

CHAPTER 9: DISCOVERING INHERITANCE TECHNIQUES

Questions about inheritance usually come in the same forms as questions about classes. The interviewer might also ask you to implement a class hierarchy to show that you have worked with C++ enough to write derived classes without looking it up in a book.

Things to Remember

- The syntax for deriving a class
- The difference between `private` and `protected` from a derived class point of view
- Method overriding and `virtual`
- What the difference is between overloading and overriding
- The reason why destructors should be `virtual`
- Chained constructors
- The ins and outs of upcasting and downcasting
- The principle of polymorphism
- Pure `virtual` methods and abstract base classes
- Multiple inheritance
- Run-Time Type Information (RTTI)
- Inherited constructors
- The `final` keyword on classes
- The `override` and `final` keywords on methods

Types of Questions

Many of the pitfalls in inheritance questions are related to getting the details right. When you are writing a base class, don't forget to make the methods `virtual`. If you mark all methods `virtual`, be prepared to justify that decision. You should be able to explain what `virtual` means and how it works. Also, don't forget the `public` keyword before the name of the parent class in the derived class definition (e.g., `class Derived : public Base`). It's unlikely that you'll be asked to perform nonpublic inheritance during an interview.

More challenging inheritance questions have to do with the relationship between a base class and a derived class. Be sure you know how the different access levels work, and the difference between `private` and `protected`. Remind yourself of the phenomenon known as *slicing*, when certain types of casts cause a class to lose its derived class information.

CHAPTER 10: C++ QUIRKS, ODDITIES, AND INCIDENTALS

Many interviewers tend to focus on the more obscure cases because that way experienced C++ programmers can demonstrate that they have conquered the unusual parts of C++. Sometimes interviewers have difficulty coming up with interesting questions and end up asking the most obscure question they can think of.

Things to Remember

- References must be bound to a variable when they are declared and the binding cannot be changed
- The advantages of pass-by-reference over pass-by-value
- The many uses of `const`
- The many uses of `static`
- The different types of casts in C++
- How `typedefs` and type aliases work
- The difference between rvalues and lvalues
- Rvalue references
- Move semantics with move constructors and move assignment operators
- Uniform initialization

Types of Questions

Asking a candidate to define `const` and `static` is a classic C++ interview question. Both keywords provide a sliding scale with which an interviewer can assess an answer. For example, a fair candidate will talk about `static` methods and `static` data members. A good candidate will give good examples of `static` methods and `static` data members. A great candidate will also know about `static` linkage and `static` variables in functions.

The edge cases described in this chapter also come in find-the-bug type problems. Be on the lookout for misuse of references. For example, imagine a class that contains a reference as a data member:

```
class Gwenyth
{
    private:
        int& mCaversham;
};
```

Because `mCaversham` is a reference, it needs to be bound to a variable when the class is constructed. To do that, you'll need to use a constructor initializer. The class could take the variable to be referenced as a parameter to the constructor:

```
class Gwenyth
{
    public:
        Gwenyth(int& i);
    private:
        int& mCaversham;
};
Gwenyth::Gwenyth(int& i) : mCaversham(i)
{}
```

CHAPTERS 11 AND 21: TEMPLATES

As one of the most arcane parts of C++, templates are a good way for interviewers to separate the C++ novices from the pros. While most interviewers will forgive you for not remembering some of the advanced template syntax, you should go into the interview knowing the basics.

Things to Remember

- How to use a class template
- How to write a basic class template
- The alternative function syntax and its use for deducing the type of the return value (type inference)
- Alias templates and why they are better than `typedefs`
- The concept of variadic templates
- The ideas behind metaprogramming

Types of Questions

Many interview questions start out with a simple problem and gradually add complexity. Often, interviewers have an endless amount of complexity that they are prepared to add, and they simply want to see how far you get. For example, an interviewer might begin a problem by asking you to create a class that provides sequential access to a fixed number of `ints`. Next, the class will need to grow to accommodate an arbitrary sized array. Then, it will need arbitrary data types, which is where templates come in. From there, the interviewer could take the problem in a number of directions, asking you to use operator overloading to provide array-like syntax, or continuing down the template path by asking you to provide a default type.

Templates are more likely to be employed in the solution of another coding problem than to be asked about explicitly. You should brush up on the basics in case the subject comes up. However, most interviewers understand that the template syntax is difficult, and asking someone to write complex template code in an interview is rather cruel.

The interviewer might ask you high-level questions related to metaprogramming to find out whether you have heard about it or not. While explaining, you could give a small example such as calculating the factorial of a number at compile time. Don't worry if the syntax is not entirely correct. As long as you explain what it is supposed to do, you should be fine.

CHAPTER 12: DEMYSTIFYING C++ I/O

If you're interviewing for a job writing GUI applications, you probably won't get too many questions about I/O streams because GUI applications tend to use other mechanisms for I/O. However, streams can come up in other problems and, as a standard part of C++, they are fair game as far as the interviewer is concerned.

Things to Remember

- The definition of a stream
- Basic input and output using streams
- The concept of manipulators
- Types of streams (console, file, string, etc.)
- Error-handling techniques

Types of Questions

I/O may come up in the context of any question. For example, the interviewer could ask you to read in a file containing test scores and put them in a vector. This question tests basic C++ skills, basic STL, and basic I/O. Even if I/O is only a small part of the problem you're working on, be sure to check for errors. If you don't, you're giving the interviewer an opportunity to say something negative about your otherwise perfect program.

CHAPTER 13: HANDLING ERRORS

Managers sometimes shy away from hiring recent graduates or novice programmers for vital (and high-paying) jobs because it is assumed that they don't write production-quality code. You can prove to an interviewer that your code won't keel over randomly by demonstrating your error-handling skills during an interview.

Things to Remember

- Syntax of exceptions
- Catch exceptions as `const` references
- For production code, hierarchies of exceptions are preferable to a few generic ones
- The basics of how stack unwinding works when an exception gets thrown
- How to handle errors in constructors and destructors
- Smart pointers help avoid memory leaks when exceptions are thrown
- Never use the C functions `setjmp()` and `longjmp()` in C++

Types of Questions

Interviewers will be on the lookout to see how you report and handle errors. When you are asked to write a piece of code, make sure you implement proper error handling.

You might be asked to give a high-level overview of how stack unwinding works when an exception is thrown, without implementation details.

Of course, not all programmers understand or appreciate exceptions. Some may even have a completely unfounded bias against them for performance reasons. If the interviewer asks you to do something without exceptions, you'll have to revert to traditional `nullptr` checks and error codes. That would be a good time to demonstrate your knowledge of `nothrow new`.

An interviewer can also ask questions in the form of “*Would you use this?*” One example question could be “*Would you use `setjmp()`/`longjmp()` in C++, since they are more efficient than exceptions?*” Your answer should be a big no, because `setjmp()`/`longjmp()` cannot possibly work in C++ because they bypass scoped destructors. The fact that exceptions have a big performance penalty is a misconception. On modern compilers, having code that can handle potential exceptions has close to zero performance penalty.

CHAPTER 14: OVERLOADING C++ OPERATORS

It's possible, though somewhat unlikely, that you would have to perform something more difficult than a simple operator overload during an interview. Some interviewers like to have an advanced question on hand that they don't really expect anybody to answer correctly. The intricacies of operator overloading make great nearly impossible questions because few programmers get the syntax right without looking it up. That means it's a great area to review before an interview.

Things to Remember

- Overloading stream operators, because they are commonly overloaded operators, and are conceptually unique
- What a functor is and how to create one
- Choosing between a method operator or a global friend function
- Some operators can be expressed in terms of others; i.e., `operator<=` can be written by complementing the result of `operator>`
- The use of rvalue references to implement move assignment operators

Types of Questions

Let's face it — operator overloading questions (other than the simple ones) can be cruel. Anybody who is asking such questions knows this and is going to be impressed when you get it right. It's impossible to predict the exact question that you'll get, but the number of operators is finite. As long as you've seen an example of overloading each operator that makes sense to overload, you'll do fine!

Besides asking you to implement an overloaded operator, you could be asked high-level questions about operator overloading. A find-the-bug question could contain an operator that is overloaded to do something that is conceptually wrong for that particular operator. In addition to syntax, keep the use cases and theory of operator overloading in mind.

CHAPTERS 15, 16, 17, AND 20: THE STANDARD TEMPLATE LIBRARY

As you've seen, certain aspects of the STL can be difficult to work with. Few interviewers would expect you to recite the details of STL classes unless you claim to be an STL expert. If you know that the job you're interviewing for makes heavy use of the STL, you might want to write some STL code the day before to refresh your memory. Otherwise, recalling the high-level design of the STL and its basic usage should suffice.

Things to Remember

- The different types of containers and their relationships with iterators
- Basic use of `vector`, which is probably the most frequently used STL class
- Use of associative containers, such as `map`
- The differences between associative containers and unordered associative containers, such as `unordered_map`
- The purpose of STL algorithms and some of the built-in algorithms
- The use of lambda expressions in combination with STL algorithms
- The ways in which you can extend the STL (details are most likely unnecessary)
- The remove-erase-idiom
- Your own opinions about the STL

Types of Questions

If interviewers are dead set on asking detailed STL questions, there really are no bounds to the types of questions they could ask. If you're feeling uncertain about syntax though, you should state the obvious during the interview — “In real life, of course, I'd look that up in *Professional C++*, but I'm pretty sure it works like this...” At least that way the interviewer is reminded that he or she should forgive the details as long as you get the basic idea right.

High-level questions about the STL are often used to gauge how much you've used the STL without making you recall all the details. For example, casual users of the STL are familiar with associative and non-associative containers. A slightly more advanced user would be able to define an iterator, describe how iterators work with containers, and describe the remove-erase-idiom. Other high-level questions could ask you about your experience with STL algorithms or whether you've customized the STL. An interviewer might also gauge your knowledge about lambda expressions, and their use with STL algorithms.

CHAPTER 19: ADDITIONAL LIBRARY UTILITIES

This chapter describes a number of smaller features and additional libraries from the C++ standard, including some C++11 and C++14 features. An interviewer might touch on a few of those topics to see whether you are keeping up-to-date with the latest developments in the C++ world.

Things to Remember

- The use of `std::function`
- The Chrono library to work with durations, clocks, and time points
- Use the `<random>` library as the preferred method of generating random numbers.
- Standard user-defined literals
- `std::tuple` as a generalization of `std::pair`

Types of Questions

You don't need to expect detailed questions about these topics. A possible question could be to explain the use of `std::function`. You might also get a question to explain the basic ideas and concepts of the Chrono and random number generation libraries, but without going into syntax details. If the interviewer starts focusing on random numbers, it is important to explain the differences between true random numbers and pseudo random numbers.

CHAPTER 22: MEMORY MANAGEMENT

You can be sure that an interviewer will ask you some questions related to memory management, including your knowledge of smart pointers. Besides smart pointers, you will also get more low-level questions. The goal is to determine whether the object-oriented aspects of C++ have distanced you too much from the underlying implementation details. Memory management questions will give you a chance to prove that you know what's really going on.

Things to Remember

- Drawing the stack and the heap can help you understand what's going on
- Use `new` and `delete` instead of `malloc()` and `free()`
- Use `new[]` and `delete[]` for arrays
- If you have an array of pointers to objects, you still need to allocate memory for each individual pointer and delete the memory — the array allocation syntax doesn't take care of pointers
- The existence of memory allocation problem detectors, such as Valgrind, to expose memory problems
- Smart pointers and specifically `std::shared_ptr` and `std::unique_ptr`, and that you should not use the old deprecated `std::auto_ptr`
- Use `std::make_unique()` to create a `std::unique_ptr`
- Use `std::make_shared()` to create a `std::shared_ptr`

Types of Questions

Find-the-bug questions often contain memory issues, such as double deletion, `new/delete/new[]` / `delete[]` mix-up, and memory leaks. When you are tracing through code that makes heavy use of pointers and arrays, you should draw and update the state of memory as you process each line of code.

Another good way to find out if a candidate understands memory is to ask how pointers and arrays differ. At this point, the differences may be so tacit in your mind that the question catches you off-guard for a moment. If that's the case, skim Chapter 22 again for the discussion.

When answering questions about memory allocation, it's always a good idea to mention the concept of smart pointers and their benefits for automatically cleaning up memory or other resources. You definitely should also mention that it's much better to use STL containers like `std::vector` instead of C-style arrays, because the STL containers handle memory management for you automatically.

CHAPTER 23: MULTITHREADED PROGRAMMING WITH C++

Multithreaded programming is becoming more and more important with the release of multicore processors for everything from servers to consumer computers. Even smartphones have multicore processors. An interviewer might ask you a couple of multithreading questions. C++ includes a standard threading library, so it's a good idea to know how it works.

Things to Remember

- Race conditions and deadlocks and how to prevent them
- `std::thread` to spawn threads
- The atomic types and atomic operations
- The concept of mutual exclusion, including the use of the different mutex and lock classes, to provide synchronization between threads
- Condition variables and how to use them to signal other threads
- Futures and promises
- Copying and rethrowing of exceptions across thread boundaries

Types of Questions

Multithreading programming is a complicated subject, so you don't need to expect detailed questions, unless you are interviewing for a specific multithreading programming position.

Instead, an interviewer might ask you to explain the different kinds of problems you can encounter with multithreaded code; Problems such as race conditions, deadlocks, tearing, and cache coherency. You can also be asked to explain the general concepts behind multithreaded programming. This is a very broad question but allows the interviewer to get an idea of your multithreading knowledge.

CHAPTER 24: MAXIMIZING SOFTWARE ENGINEERING METHODS

You should be suspicious if you go through the complete interview process with a company, and the interviewers do *not* ask any process questions — it may mean that they don't have any process or that they don't care about it. Alternatively, they might not want to scare you away with their process behemoth. Another important aspect of any development process is source code control.

Most of the time, you'll get a chance to ask questions regarding the company. I suggest you consider asking about engineering processes and source code control solutions as one of your standard questions.

Things to Remember

- Traditional life-cycle models
- The tradeoffs of formal models, such as the Rational Unified Process
- The main principles of Extreme Programming
- Scrum as an example of an agile process
- Other processes you have used in the past
- Principles behind source code control solutions

Types of Questions

The most common question you'll be asked is to describe the process that your previous employer used. When answering, you should mention what worked well and what failed, but try not to denounce any particular methodology. The methodology you criticize could be the one that your interviewer uses.

Almost every candidate is listing Extreme Programming as a skill these days. While there's little hard data on the subject, it certainly seems unlikely that strict adherence to XP is commonplace in programming environments. It is more likely that many organizations have started to look into XP and have adopted some of its principles without subscribing to it in any formal way.

If the interviewer asks you about XP, he or she probably doesn't want you to simply recite the textbook definition — the interviewer knows that you can read the table of contents of an XP book. Instead, pick a few ideas from XP that you find appealing. Explain each to the interviewer along with your thoughts on them. Try to engage the interviewer in a conversation, proceeding in a direction in which he or she is interested based on the cues that person gives.

If you get a question regarding source code control it will most likely be a high-level question. You should explain the general principles behind source code control solutions, mention the fact that there are commercial and free open-source solutions available, and possibly explain how source code control happened at your previous employer.

CHAPTER 25: WRITING EFFICIENT C++

Efficiency questions are quite common in interviews because many organizations are facing scalability issues with their code and need programmers who are savvy about performance.

Things to Remember

- Language-level efficiency is important, but it can only go so far; Design-level choices are ultimately much more significant.
- Avoid algorithms with bad complexity, such as quadratic algorithms.
- Reference parameters are more efficient because they avoid copying.
- Object pools can help avoid the overhead of creating and destroying objects.
- Profiling is vital to determine which operations are really consuming the most time, so you don't waste effort trying to optimize code that is not a performance bottleneck.

Types of Questions

Often, the interviewer will use his or her own product as an example to drive efficiency questions. Sometimes the interviewer will describe an older design and some performance-related symptoms he or she experienced. The candidate is supposed to come up with a new design that alleviates the problem. Unfortunately, there is a major problem with a question like this — what are the odds that you're going to come up with the same solution that the company did when the problem was actually solved? Because the odds are slim, you need to be extra careful to justify your designs. You may not come up with the actual solution, but you can still have an answer that is correct or even better than the company's newer design.

Other types of efficiency questions may ask you to tweak some C++ code for performance or iterate on an algorithm. For example, the interviewer could show you code that contains extraneous copies or inefficient loops.

The interviewer might also ask you for a high-level description of profiling tools and what their benefits are.

CHAPTER 26: CONQUERING DEBUGGING

Engineering organizations look for candidates who are able to debug both their own code as well as code that they've never seen before. Technical interviews often attempt to size up your debugging muscles.

Things to Remember

- Debugging doesn't start when bugs appear; you should instrument your code ahead of time, so you're prepared for bugs when they arrive
- Logs and debuggers are your best tools

- How to use assertions
- How to use static assertions
- The symptoms that a bug exhibits may appear to be unrelated to the actual cause
- Object diagrams can be helpful in debugging, especially during an interview

Types of Questions

During an interview, you might be challenged with an obscure debugging problem. Remember that the process is the most important thing, and the interviewer probably knows that. Even if you don't find the bug during the interview, make sure that the interviewer knows what steps you would go through to track it down. If the interviewer hands you a function and tells you that it crashes during execution, he or she should award just as many points to a candidate who properly discusses the sequence of steps to find the bug, as to a candidate who finds it right away.

B

Annotated Bibliography

This appendix contains a list of books and online resources on various C++-related topics that were either consulted while writing this book, or are recommended for further or background reading.

C++

Beginning C++

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, *C++ Primer (5th Edition)*, Addison-Wesley, 2012, ISBN: 0-321-71411-3.

A very thorough introduction to C++ that covers just about everything in the language in a very accessible format and in great detail.

- Bruce Eckel, *Thinking in C++, Volume 1: Introduction to Standard C++ (Second Edition)*, Prentice Hall, 2000, ISBN: 0-139-79809-9.

An excellent introduction to C++ programming that expects the reader to know C already. Available at no cost online at www.bruceeckel.com.

- Andrew Koenig, Barbara E. Moo, *Accelerated C++: Practical Programming by Example*, Addison-Wesley Professional, 2000, ISBN: 0-201-70353-X.

Covers the same material as C++ Primer, but in much less space, because it assumes that the reader has programmed in another language before.

- Steve Oualline, *Practical C++ Programming (Second Edition)*, O'Reilly, 2003, ISBN: 0-596-00419-2.

An introductory C++ text that assumes no prior programming experience.

- Walter Savitch, *Problem Solving with C++ (Eighth Edition)*, Addison-Wesley, 2011, ISBN: 0-132-16273-3.

This book assumes no prior programming experience. It is often used as a textbook in introductory programming courses.

General C++

- The C++ Programming Language at www.isocpp.org.

The home of Standard C++ on the web — news, status and discussions about the C++ standard on all compilers and platforms.

- Marshall Cline, *C++ FAQ LITE*, www.parashift.com/c++-faq-lite.
- Marshall Cline, Greg Lomow, and Mike Girou, *C++ FAQs (Second Edition)*, Addison-Wesley, 1998, ISBN: 0-201-30983-1.

This compilation of frequently asked questions from the `comp.lang.c++` newsgroup is useful for quickly looking up a specific point about C++. The printed version contains more information than the online version, but the material available online should be sufficient for most professional C++ programmers.

- Stephen C. Dewhurst, *C++ Gotchas*, Addison-Wesley, 2002, ISBN: 0-321-12518-5.
Provides 99 specific tips for C++ programming.
- Bruce Eckel and Chuck Allison, *Thinking in C++, Volume 2: Practical Programming*, Prentice Hall, 2003, ISBN: 0-130-35313-2.

The second volume of Eckel's book covers more advanced C++ topics. It's also available at no cost online at www.bruceeckel.com.

- Ray Lischner, *C++ in a Nutshell*, O'Reilly, 2003, ISBN: 0-596-00298-X.
A C++ reference covering everything from the basics to more-advanced material.
- Scott Meyers, *Effective C++ (Third Edition): 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 2005, ISBN: 0-321-33487-6.
- Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, ISBN: 0-201-63371-X.

These two books provide excellent tips and tricks on commonly misused and misunderstood features of C++.

- Stephen Prata, *C++ Primer Plus (Fifth Edition)*, Sams Publishing, 2004, ISBN: 0-672-32697-4.
One of the most comprehensive C++ books available.
- Bjarne Stroustrup, *The C++ Programming Language (Special Third Edition)*, Addison-Wesley, 2000, ISBN: 0-201-70073-5.

The “Bible” of C++ books, written by the designer of C++ himself. Every C++ programmer should own a copy of this book, but it can be a bit obscure in places for the C++ novice.

- Newsgroups at <http://groups.google.com>, including `comp.lang.c++.moderated` and `comp.std.c++`.

The newsgroups contain a lot of useful information if you're willing to wade through the flame wars, insults, and misinformation that appear as well.

- The C++ Reference at www.cppreference.com.
An excellent reference of C++98, C++03, C++11, and C++14.
- *The C++ Resources Network* at www.cplusplus.com.
This site contains a lot of information related to C++, including a complete reference of the language.

I/O Streams and Strings

- Cameron Hughes and Tracey Hughes, *Mastering the Standard C++ Classes: An Essential Reference*, Wiley, 1999, ISBN: 0-471-32893-6.
A good book for learning how to write custom `istream` and `ostream` classes.
- Cameron Hughes and Tracey Hughes, *Stream Manipulators and Iterators in C++*, www.informit.com/articles/article.aspx?p=171014.
This well-written article by the authors of *Mastering the Standard C++ Classes* takes the mystery out of defining custom stream manipulators in C++.
- Philip Romanik and Amy Muntz, *Applied C++: Practical Techniques for Building Better Software*, Addison-Wesley, 2003, ISBN: 0-321-10894-9.
In addition to a unique blend of software development advice and C++ specifics, this book provides a very good explanation of locale and Unicode support in C++.
- Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, www.joelonsoftware.com/articles/Unicode.html.
After reading Joel's treatise on the importance of localization, you'll want to check out his other entries on *Joel on Software*.
- The Unicode Consortium, *The Unicode Standard 5.0*, Addison-Wesley, 2006, ISBN: 0-321-48091-0.
This is the definitive book on Unicode, which all developers using Unicode must have.
- Unicode, Inc., *Where is my Character?*, www.unicode.org/standard/where.
The best resource for finding Unicode characters, charts, and tables.
- *Wikipedia Universal Character Set*, http://en.wikipedia.org/wiki/Universal_Character_Set.
An explanation of what the Universal Character Set (UCS) is, including the Unicode standard.

The C++ Standard Library

- Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999, ISBN: 0-201-37926-0.
This book covers the entire standard library, including I/O streams and strings as well as the containers and algorithms. It's an excellent reference.

- Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, 2001, ISBN: 0-201-74962-9.

Meyers wrote this book in the same spirit as his “*Effective C++*” books. It provides targeted tips for using the STL, but is not a reference or tutorial.
- David R. Musser, Gillmer J. Derge, and Atul Saini, *STL Tutorial and Reference Guide (Second Edition)*, Addison-Wesley, 2001, ISBN: 0-321-70212-3.

This book is similar to the Josuttis text, but covers only the STL part of the standard library.
- Pete Becker, *The C++ Standard Library Extensions: A Tutorial and Reference*, Addison-Wesley, 2006, ISBN: 0-321-41299-0.

This book explains the new features added to the C++ Standard Library with the Technical Report 1 (TR1).
- Stephan T. Lavavej, *Standard Template Library (STL)*, <http://channel9.msdn.com/Shows/Going+Deep/C9-Lectures-Introduction-to-STL-with-Stephan-T-Lavavej>.

An interesting video lecture series on the C++ Standard Template Library.

C++ Templates

- Herb Sutter, *Sutter’s Mill: Befriending Templates*, C/C++ User’s Journal, <http://drdobbs.com/cpp/184403853>.

An excellent explanation about making function templates friends of classes.
- David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2002, ISBN: 0-201-73484-2.

Everything you ever wanted to know (or didn’t want to know) about C++ templates. It assumes significant background in general C++.
- David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, 2004, ISBN: 0-321-22725-5.

This book delivers practical metaprogramming tools and techniques into the hands of the everyday programmer.

C++11 / C++14

- *C++ Standards Committee Papers*, www.open-std.org/jtc1/sc22/wg21/docs/papers.

Access a wealth of papers written by the C++ standards committee.
- Scott Meyers, *Presentation Materials: Overview of the New C++ (C++11/C++14)*, Artima, 2013, www.artima.com/shop/overview_of_the_new_cpp.

This contains the presentation materials from Scott Meyers’ training course on the new C++ standard, and is an excellent reference to get a list of all new C++11 features and select C++14 features.

- *Wikipedia C++11*, <http://en.wikipedia.org/wiki/C%2B%2B11>.
A description of all new features added to C++11.
- *Wikipedia C++14*, <http://en.wikipedia.org/wiki/C%2B%2B14>.
A description of a selection of new features added to C++14.
- *ECMAScript Language Specification*, www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf.
One of the syntaxes of the regular expressions in C++ is the same as the regular expressions in the ECMAScript language, described in this specification document.

C

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language (second edition)*, Prentice Hall, 1998, ISBN: 0-131-10362-8.
“K&R,” as this book is known, is a reference on the C language, but it’s not so useful for learning it the first time.
- Samuel P. Harbison III and Guy L. Steele Jr., *C: A Reference Manual, Fifth Edition*, Prentice Hall, 2002, ISBN: 0-130-89592-X.
This book can be considered a replacement for the K&R book. Instead of a narrative style, where knowledge is embedded in the text, it is done as a more formal reference manual. Every C programmer should own a copy.
- Peter Prinz, Tony Crawford (Translator), Ulla Kirch-Prinz, *C Pocket Reference*, O’Reilly, 2002, ISBN: 0-596-00436-2.
A concise reference to all things C.
- Eric S. Roberts, *The Art and Science of C: A Library Based Introduction to Computer Science*, Addison-Wesley, 1994, ISBN: 0-201-54322-2.
- Eric S. Roberts, *Programming Abstractions in C: A Second Course in Computer Science*, Addison-Wesley, 1997, ISBN: 0-201-54541-1.
These two books provide a great introduction to programming in C with good style. They are often used as textbooks in introductory programming courses.
- Peter Van Der Linden, *Expert C Programming: Deep C Secrets*, Prentice Hall, 1994, ISBN: 0-131-77429-8.
An enlightening and often hysterical look at the C language, its evolution, and its inner workings.

UNIFIED MODELING LANGUAGE, UML

- Russ Miles, and Kim Hamilton, *Learning UML 2.0*, O’Reilly Media, 2006, ISBN: 0-596-00982-8.
A very readable book on UML 2.0. It uses Java in examples, but those are convertible to C++ without too much trouble.

ALGORITHMS AND DATA STRUCTURES

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms (Third Edition)*, The MIT Press, 2009, ISBN: 0-262-03384-4.
This text is one of the most popular introductory algorithms books, covering all the common data structures and algorithms.
- Donald E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms (Third Edition)*, Addison-Wesley, 1997, ISBN: 0-201-89683-4.
- Donald E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms (Third Edition)*, Addison-Wesley, 1997, ISBN: 0-201-89684-2.
- Donald E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching (Second Edition)*, Addison-Wesley, 1998, ISBN: 0-201-89685-0.
- Donald E. Knuth, *The Art of Computer Programming Volume 4A: Combinatorial Algorithms Part 1*, Addison-Wesley, 2011, ISBN: 0-201-03804-8.
For those of you who enjoy mathematical rigor, there is no better algorithms and data structures text than Knuth's four-volume tome. But, it is probably inaccessible without undergraduate knowledge of mathematics or theoretical computer science.
- Kyle Loudon, *Mastering Algorithms with C*, O'Reilly, 1999, ISBN: 1-565-92453-3.
An approachable reference to data structures and algorithms.

RANDOM NUMBERS

- Eric Bach and Jeffrey Shallit, *Algorithmic Number Theory, Vol. 1: Efficient Algorithms*, The MIT Press, 1996, ISBN: 0-262-02405-5.
- Oded Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*, Springer, 2010, ISBN: 3-642-08432-X.
Both of these books explain the theory of computational pseudo randomness.
- *Wikipedia Mersenne Twister*, http://en.wikipedia.org/wiki/Mersenne_twister.
A mathematical explanation of the Mersenne Twister to generate pseudo-random numbers.

OPEN-SOURCE SOFTWARE

- The Open Source Initiative at www.opensource.org.
 - The GNU Operating System — Free Software Foundation at www.gnu.org.
- The two main open-source movements explain their philosophies and provide information about obtaining open-source software and contributing to its development.

- The Boost C++ Libraries at www.boost.org.
Boost provides a huge amount of free peer-reviewed portable C++ source libraries. Definitely worth checking out.
- SourceForge at www.sourceforge.net, and GitHub at www.github.com.
These websites hosts many open-source projects. It's a great resource for finding useful open-source software.
- www.codeguru.com and www.codeproject.com.
Excellent resources to find free libraries and code for reuse in your own projects.

SOFTWARE ENGINEERING METHODOLOGY

- Barry W. Boehm, TRW Defense Systems Group, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21(5): 61–72, 1988.
This landmark paper described the state of software development at the time and proposed the Spiral Model.
- Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (Second Edition)*, Addison-Wesley, 2004, ISBN: 0-321-27865-8.
One of several books in a series that promote Extreme Programming as a new approach to software development.
- Robert T. Futrell, Donald F. Shafer, and Linda Isabell Shafer, *Quality Software Project Management*, Prentice Hall, 2002, ISBN: 0-130-91297-2.
A guidebook for anybody who is responsible for the management of software development processes.
- Robert L. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2002, ISBN: 0-321-11742-5.
This book discusses various aspects of the software development process and exposes hidden truisms along the way.
- Philippe Kruchten, *The Rational Unified Process: An Introduction (Third Edition)*, Addison-Wesley, 2003, ISBN: 0-321-19770-4.
Provides an overview of RUP, including its mission and processes.
- Edward Yourdon, *Death March (Second Edition)*, Prentice Hall, 2003, ISBN: 0-131-43635-X.
A wonderfully enlightening book about the politics and realities of software development.
- Rational Unified Process from IBM, www3.software.ibm.com/ibmdl/pub/software/rational/web/demos/viewlets/rup/runtime/index.html.
The IBM website contains a wealth of information about RUP, including the interactive presentation at the preceding URL.

- Mike Cohn, *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley, 2009, ISBN: 0-321-57936-4.
An excellent guide to start with the Scrum methodology.
- *Wikipedia Scrum*, [http://en.wikipedia.org/wiki/Scrum_\(software_development\)](http://en.wikipedia.org/wiki/Scrum_(software_development)).
A detailed discussion of the Scrum methodology.
- *Manifesto for Agile Software Development*, <http://agilemanifesto.org/>.
The complete agile software development manifesto.
- *Wikipedia Revision control*, http://en.wikipedia.org/wiki/Revision_control.
Explains the concepts behind revision control systems, and what kinds of solutions there are available.

PROGRAMMING STYLE

- Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999, ISBN: 0-201-48567-2.
This classic book espouses the practice of recognizing and improving bad code.
- Herb Sutter, and Andrei Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley, 2004, ISBN: 0-321-11358-6.
A must-have book on C++ design and coding style.
- Diomidis Spinellis, *Code Reading: The Open Source Perspective*, Addison-Wesley, 2003, ISBN: 0-201-79940-5.
This unique book turns the issue of programming style upside down by challenging the reader to learn to read code properly in order to become a better programmer.
- Dimitri van Heesch, *Doxygen*, www.stack.nl/~dimitri/doxygen/index.html.
A highly configurable program that generates documentation from source code and comments.
- John Ayccock, *Reading and Modifying Code*, John Ayccock, 2008, ISBN 0-980-95550-5.
A nice little book with advice about how to perform the most common operations on code: reading, modifying, testing, debugging, and writing.
- *Wikipedia Code Refactoring*, <http://en.wikipedia.org/wiki/Refactoring>.
A discussion on what code refactoring means, including a number of techniques for refactoring.

COMPUTER ARCHITECTURE

- David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface (Fourth Edition)*, Morgan Kaufmann, 2008, ISBN: 0-123-74493-8.

-
- John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach (Fourth Edition)*, Morgan Kaufmann, 2006, ISBN: 0-123-70490-1.

These two books provide all the information most software engineers ever need to know about computer architecture.

EFFICIENCY

- Dov Bulka and David Mayhew, *Efficient C++: Performance Programming Techniques*, Addison-Wesley, 1999, ISBN: 0-201-37950-3.

One of the few books to focus exclusively on efficient C++ programming, it covers both language-level and design-level efficiency.

- GNU gprof, www.gnu.org/software/binutils/.

Information about the gprof profiling tool.

TESTING

- Elfriede Dustin, *Effective Software Testing: 50 Specific Ways to Improve Your Testing*, Addison-Wesley, 2002, ISBN: 0-201-79429-2.

While this book is aimed at quality assurance professionals, any software engineer will benefit from its discussion of the software-testing process.

DEBUGGING

- The GNU DeBugger (GDB), at www.gnu.org/software/gdb/gdb.html.

GDB is an excellent symbolic debugger.

- Valgrind, at <http://valgrind.org/>.

An open-source memory-debugging tool for Linux.

- Microsoft Application Verifier, at <http://msdn.microsoft.com/en-us/library/aa480483.aspx>.

A run-time verification tool for C++ code that assists in finding subtle programming errors and security issues that can be difficult to identify with normal application testing techniques.

DESIGN PATTERNS

- Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001, ISBN: 0-201-70431-5.

Offers an approach to C++ programming employing highly reusable code and patterns.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, ISBN: 0-201-63361-2.

Called the “Gang of Four” (GoF) book (because of its four authors), this text is the seminal work on design patterns.
- John Vlissides, *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998, ISBN: 0-201-43293-5.

A companion to the GoF book explaining how patterns can actually be applied.
- Eric Freeman, Bert Bates, Kathy Sierra, and Elisabeth Robson, *Head First Design Patterns*, O'Reilly Media, 2004, ISBN: 0-596-00712-4.

A book that goes further than just listing design patterns. It shows good and bad examples of using patterns, and gives solid reasoning behind each pattern.
- *Wikipedia Design Patterns*, [http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)).

Contains a description of a large number of design patterns used in computer programming.

OPERATING SYSTEMS

- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating System Concepts (Eighth Edition)*, Wiley, 2008, ISBN: 0-470-12872-0.

A great discussion on operating systems, including multithreading issues such as deadlocks and race conditions.

MULTITHREADED PROGRAMMING

- Anthony Williams, *C++ Concurrency in Action: Practical Multithreading*, Manning Publications, 2011, ISBN: 1-933-98877-0.

An excellent book on practical multithreaded programming, including the C++ threading library.
- Cameron Hughes and Tracey Hughes, *Professional Multicore Programming: Design and Implementation for C++ Developers*, Wrox, 2008, ISBN: 0-470-28962-7.

This book is for developers of various skill levels who are making the move into multicore programming.
- Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008, ISBN: 0-123-70591-6.

A great book on writing code for multiprocessor and multicore systems.
- *Wikipedia POSIX Threads*, http://en.wikipedia.org/wiki/POSIX_Threads.
- *Boost Threads*, www.boost.org.

Explains how to work with POSIX threads or Boost threads in case your compiler does not yet support the C++ threading library.

C

Standard Library Header Files

The interface to the C++ Standard Library consists of 80 header files, 26 of which present the C standard library. It's often difficult to remember which header files you need to include in your source code, so this material provides a brief description of the contents of each header, organized into eight categories:

- The C Standard Library
- Containers
- Algorithms, iterators, and allocators
- General utilities
- Mathematical utilities
- Exceptions
- I/O Streams
- Threading library

THE C STANDARD LIBRARY

The C++ Standard Library includes the entire C Standard Library. The header files are generally the same, except for two points:

- The header names are `<cname>` instead of `<name.h>`.
- All the names declared in the `<cname>` header files are in the `std` namespace.

NOTE *For backward compatibility, you can still include `<name.h>` if you want. However, that puts the names into the global namespace instead of the `std` namespace, and on top of that, the use of `<name.h>` has been deprecated. It is recommended to avoid this feature.*

The following table provides a summary of the most useful functionality. Note that it's recommended to avoid using C functionality, and instead use equivalent C++ features whenever possible.

HEADER FILE NAME	CONTENTS
<code><cassert></code>	<code>assert()</code> macro.
<code><ccomplex></code>	Utilities to work with complex numbers.
<code><cctype></code>	Character predicates and manipulation functions, such as <code>isspace()</code> and <code>tolower()</code> .
<code><cerrno></code>	Defines <code>errno</code> expression, a macro to get the last error number for certain C functions.
<code><cfenv></code>	Supports the floating-point environment, such as floating-point exceptions, rounding, and so on.
<code><cfloat></code>	C-style defines related to floating-point arithmetic, such as <code>FLT_MAX</code> .
<code><cinttypes></code>	Defines a number of macros to use with the <code>printf()</code> , <code>scanf()</code> and similar functions. Also includes a few functions to work with <code>intmax_t</code> .
<code><ciso646></code>	In C, the <code><iso646.h></code> file defines macros and, or, etc. In C++, those are keywords, so this header is empty.
<code><climits></code>	C-style limit defines, such as <code>INT_MAX</code> .
<code><locale></code>	A few localization macros and functions like <code>LC_ALL</code> and <code>setlocale()</code> .
<code><cmath></code>	Math utilities, including trigonometric functions, <code>sqrt()</code> , <code>fabs()</code> , and others.
<code><csetjmp></code>	<code>setjmp()</code> and <code>longjmp()</code> . Never use these in C++!
<code><csignal></code>	<code>signal()</code> and <code>raise()</code> . Avoid these in C++.
<code><cstdalign></code>	Alignment related macro <code>__alignas_is_defined</code> .
<code><cstdarg></code>	Macros and types for processing variable-length argument lists.
<code><cstdbool></code>	Boolean type related macro <code>__bool_true_false_are_defined</code> .
<code><cstddef></code>	Important constants such as <code>NULL</code> , and important types such as <code>size_t</code> .
<code><cstdint></code>	Defines a number of standard integer types such as <code>int8_t</code> , <code>int64_t</code> and so on. It also includes macros specifying minimum and maximum values of those types.
<code><cstdio></code>	File operations, including <code>fopen()</code> and <code>fclose()</code> . Formatted I/O: <code>printf()</code> , <code>scanf()</code> , and family. Character I/O: <code>getc()</code> , <code>putc()</code> , and family. File positioning: <code>fseek()</code> , <code>ftell()</code> .

HEADER FILE NAME	CONTENTS
<code><cstdlib></code>	Random numbers with <code>rand()</code> and <code>rand()</code> . The <code>abort()</code> and <code>exit()</code> functions, which you should avoid. C-style memory allocation functions: <code>calloc()</code> , <code>malloc()</code> , <code>realloc()</code> , <code>free()</code> . C-style searching and sorting with <code>qsort()</code> and <code>bsearch()</code> . String to number conversions: <code>atof()</code> , <code>atoi()</code> , etc. A set of functions related to multibyte/wide strings manipulation.
<code><cstring></code>	Low-level memory management functions, including <code>memcpy()</code> and <code>memset()</code> . C-style string functions, such as <code>strcpy()</code> and <code>strcmp()</code> .
<code><ctgmath></code>	Just includes <code><ccomplex></code> and <code><cmath></code> .
<code><ctime></code>	Time-related functions, including <code>time()</code> and <code>localtime()</code> .
<code><cuchar></code>	Defines a number of Unicode-related macros, and functions like <code>mbrtoc16()</code> .
<code><cwchar></code>	Versions of string, memory, and I/O functions for wide characters.
<code><cwctype></code>	Versions of functions in <code><cctype></code> for wide characters: <code>iswspace()</code> , <code>towlower()</code> , and so on.

CONTAINERS

The definitions for the STL containers can be found in 12 header files:

HEADER FILE NAME	CONTENTS
<code><array></code>	The <code>array</code> class template.
<code><bitset></code>	The <code>bitset</code> class template.
<code><deque></code>	The <code>deque</code> class template.
<code><forward_list></code>	The <code>forward_list</code> class template.
<code><list></code>	The <code>list</code> class template.
<code><map></code>	The <code>map</code> and <code>multimap</code> class templates.
<code><queue></code>	The <code>queue</code> and <code>priority_queue</code> class templates.
<code><set></code>	The <code>set</code> and <code>multiset</code> class templates.
<code><stack></code>	The <code>stack</code> class template.
<code><unordered_map></code>	The <code>unordered_map</code> and <code>unordered_multimap</code> class templates.
<code><unordered_set></code>	The <code>unordered_set</code> and <code>unordered_multiset</code> class templates.
<code><vector></code>	The <code>vector</code> class template and the <code>vector<bool></code> specialization.

Each of these header files contains all the definitions you need to use the specified container, including iterators. Chapter 16 describes these containers in detail.

ALGORITHMS, ITERATORS, AND ALLOCATORS

The “rest” of the STL can be found in six different header files:

HEADER FILE NAME	CONTENTS
<code><algorithm></code>	Prototypes for most of the algorithms in the STL. See Chapter 17.
<code><functional></code>	Defines the built-in function objects, negators, binders, and adaptors. See Chapter 17.
<code><iterator></code>	Definitions of <code>iterator_traits</code> , <code>iterator tags</code> , <code>iterator</code> , <code>reverse_iterator</code> , <code>insert iterators</code> (such as <code>back_inserter</code>), and <code>stream iterators</code> . See Chapter 20.
<code><memory></code>	Defines the default allocator, functions for dealing with uninitialized memory inside containers, <code>unique_ptr</code> , <code>shared_ptr</code> , <code>make_unique()</code> , and <code>make_shared()</code> introduced in Chapter 1.
<code><numeric></code>	Prototypes for some numerical algorithms: <code>accumulate()</code> , <code>inner_product()</code> , <code>partial_sum()</code> , <code>adjacent_difference()</code> , and a few others. See Chapter 17.
<code><scoped_allocator></code>	An allocator that can be used with nested containers such as a <code>vector</code> of <code>strings</code> , or a <code>vector</code> of <code>maps</code> .

GENERAL UTILITIES

The Standard Library contains some general-purpose utilities in several different header files:

HEADER FILE NAME	CONTENTS
<code><chrono></code>	Defines the Chrono library. See Chapter 19.
<code><codecvt></code>	Provides code conversion facets for various character encodings.
<code><initializer_list></code>	Defines the <code>initializer_list</code> class. See Chapter 10.
<code><limits></code>	Defines the <code>numeric_limits</code> class template, and specializations for most built-in types. See Chapter 15.
<code><locale></code>	Defines the <code>locale</code> class, the <code>use_facet()</code> and <code>has_facet()</code> function templates, and the various facet families. See Chapter 18.
<code><new></code>	Defines the <code>bad_alloc</code> exception and <code>set_new_handler()</code> function. Prototypes for all six forms of <code>operator new</code> and <code>operator delete</code> . See Chapter 14.
<code><random></code>	Defines the random number generation library. See Chapter 19.

HEADER FILE NAME	CONTENTS
<code><ratio></code>	Defines the <code>Ratio</code> library to work with compile-time rational numbers. See Chapter 19.
<code><regex></code>	Defines the regular expression library. See Chapter 18.
<code><string></code>	Defines the <code>basic_string</code> class template and the <code>typedef</code> instantiations of <code>string</code> and <code>wstring</code> .
<code><system_error></code>	Defines error categories and error codes.
<code><tuple></code>	Defines the <code>tuple</code> class template as a generalization of the <code>pair</code> class template. See Chapter 19.
<code><type_traits></code>	Defines type traits for use in template metaprogramming. See Chapter 21.
<code><typeindex></code>	Defines a simple wrapper for <code>type_info</code> , which can be used as an index type in associative containers and in unordered associative containers.
<code><typeinfo></code>	Defines the <code>bad_cast</code> and <code>bad_typeid</code> exceptions. Defines the <code>type_info</code> class, objects of which are returned by the <code>typeid</code> operator. See Chapter 9 for details on <code>typeid</code> .
<code><utility></code>	Defines the <code>pair</code> class template. See Chapter 16.

MATHEMATICAL UTILITIES

C++ provides some facilities for numeric processing. These capabilities are not described in detail in this book; for details, consult one of the Standard Library references listed in the Annotated Bibliography.

HEADER FILE NAME	CONTENTS
<code><complex></code>	Defines the <code>complex</code> class template for working with complex numbers.
<code><valarray></code>	Defines <code>valarray</code> and related classes and class templates for working with mathematical vectors and matrices.

EXCEPTIONS

Exceptions and exception support are covered in Chapter 13. Two header files provide most of the requisite definitions, but some exceptions for other domains are defined in the header file for that domain.

HEADER FILE NAME	CONTENTS
<code><exception></code>	Defines the <code>exception</code> and <code>bad_exception</code> classes, and the <code>set_unexpected()</code> , <code>set_terminate()</code> , and <code>uncaught_exception()</code> functions.
<code><stdexcept></code>	Non-domain-specific exceptions not defined in <code><exception></code> .

I/O STREAMS

The following table lists all the header files related to I/O streams in C++. However, normally your applications only need to include `<fstream>`, `<iomanip>`, `<iostream>`, `<iostream>`, `<ostream>`, and `<sstream>`. Consult Chapter 12 for details.

HEADER FILE NAME	CONTENTS
<code><fstream></code>	Defines the <code>basic_filebuf</code> , <code>basic_ifstream</code> , <code>basic_ofstream</code> , and <code>basic_fstream</code> classes. Declares the <code>filebuf</code> , <code>wfilebuf</code> , <code>ifstream</code> , <code>wifstream</code> , <code>ofstream</code> , <code>wofstream</code> , <code>fstream</code> , and <code>wfstream</code> typedefs.
<code><iomanip></code>	Declares the I/O manipulators not declared elsewhere (mostly in <code><ios></code>).
<code><ios></code>	Defines the <code>ios_base</code> and <code>basic_ios</code> classes. Declares most of the stream manipulators. You rarely have to include this header directly.
<code><iosfwd></code>	Forward declarations of the templates and <code>typedefs</code> found in the other I/O stream header files. You rarely need to include this header directly.
<code><iostream></code>	Declares <code>cin</code> , <code>cout</code> , <code>cerr</code> , <code>clog</code> , and the wide-character counterparts. Note that it's not just a combination of <code><iostream></code> and <code><ostream></code> .
<code><iostream></code>	Defines the <code>basic_istream</code> and <code>basic_iostream</code> classes. Declares the <code>istream</code> , <code>wistream</code> , <code>iostream</code> , and <code>wiostream</code> typedefs.
<code><ostream></code>	Defines the <code>basic_ostream</code> class. Declares the <code>ostream</code> and <code>wostream</code> typedefs.
<code><sstream></code>	Defines the <code>basic_stringbuf</code> , <code>basic_istringstream</code> , <code>basic_ostringstream</code> , and <code>basic_stringstream</code> classes. Declares the <code>stringbuf</code> , <code>wstringbuf</code> , <code>istringstream</code> , <code>wistringstream</code> , <code>ostringstream</code> , <code>wostringstream</code> , <code>stringstream</code> , and <code>wstringstream</code> typedefs.
<code><streambuf></code>	Defines the <code>basic_streambuf</code> class. Declares the <code>typedefs</code> <code>streambuf</code> and <code>wstreambuf</code> . You rarely have to include this header directly.
<code><strstream></code>	Deprecated.

THREADING LIBRARY

C++ includes a threading library, which allows you to write platform-independent multithreaded applications. See Chapter 23 for details. The threading library consists of the following header files:

HEADER FILE NAME	CONTENTS
<code><atomic></code>	Defines the atomic types, <code>atomic<T></code> , and atomic operations.
<code><condition_variable></code>	Defines the <code>condition_variable</code> and <code>condition_variable_any</code> classes.
<code><future></code>	Defines <code>future</code> , <code>promise</code> , <code>packaged_task</code> and <code>async()</code> .
<code><mutex></code>	Defines <code>call_once()</code> and the different mutex and lock classes, except <code>shared_timed_mutex</code> and <code>shared_lock</code> .
<code><shared_mutex></code>	Defines the <code>shared_timed_mutex</code> and <code>shared_lock</code> classes.
<code><thread></code>	Defines the <code>thread</code> class.

INDEX

Numbers & Symbols

= operator, 12
/ (backslash character), 7
* (quotation mark), 7
n (escape character), 6, 7
r (carriage return), 7
t (tab), 7
- - decrement operator, 12
+ addition operator, 12
/ division operator, 12
~ in destructors, 180
in directives, 5
++ increment operator, 12
% mod operator, 12
* multiplication operator, 12
! operator, 11
%<= operator, 12
& operator, 12
&=< operator, 12
*< operator, 12
+< operator, 12
-< operator, 12
/=< operator, 12
<< operator, 6, 12
= operator, 11
>> operator, 12
>>=< operator, 12
^ operator, 12
^=< operator, 12
| operator, 12
|= operator, 12
+ operator, string class, 51
- subtraction operator, 12

16-bit characters, 10
32-bit characters, 10

A

abstract base classes, 238–239
abstract superclasses, 115
abstraction, 122, 211, 212
 benefits, 84
 classes, 130
 design and, 84, 85
 design success, 125
 interface
 exposed, 123, 125
 versus implementation, 123
 reusable code, 129, 130
access specifiers, 145, 146
 private, 146
 protected, 146
 public, 146
accumulate function, 535–536, 570
ad hoc comments, 65
adapters
 containers
 example, 499–504
 operations, 499, 502–503
 priority_queue, 501–503
 queue, 498–501
 function objects
 bind2nd(), 548
 std::bind(), 546–548
add() method, overloading, 203–204
addition operator (+), 12
addOne() function, 30

aggregation, 113–114
 reusable code, 132

Agile Methodology, 789

algorithms. *See also* functions

- `binary_search()`, 566
- C++ standard library, 443
- callbacks, 544
- comparison algorithms, 553–555
- `count_if()`, 542
- `equal_range()`, 566
- examples, voter registration audits, 570–573
- extending STL, 634–638
- `for_each()`, 562
- `generate()`, 542
- header files, 898
- interface, 448
- iterator invalidation and, 485
- iterators, 551
- `lower_bound()`, 566
- modifying algorithms, 556–557
 - `copy()`, 558–559
 - `erase()`, 561–562
 - `iota()`, 570
 - `move()`, 559–560
 - `replace()`, 560–561
 - `reverse()`, 562
 - `transform()`, 556–557
 - `unique()`, 562
- move semantics, 536
- non-modifying, search algorithms, 551–553
- numerical processing, 570
- operational algorithms, 562–564
- overview, 532–536
- set algorithms, 566–569
- sorting algorithms, 65–566
- STL (standard template library), 456–458
 - comparison, 459
 - heap, 463
 - modifying, 459–460
 - non-modifying, 458–459
 - numerical processing, 464
 - operational, 461

search algorithms, 458

- sorting, 462
- utility algorithms, 459

- `upper_bound()`, 566
- utility, 555–556

aliasing, 728

- alias templates, 338
- type aliases, 297

alignment, curly braces, 72–73

`allocate()` method, 628

allocation operators, 436–442

`Allocator` parameter, 473

`Allocator` type, 628

allocators, 657

- header files, 898

`allTrue()` function, 545

alternation (ECMAScript), 582

alternative function syntax, 23, 338

ambiguity in naming, 246–247

- base classes, 248

analysis and design workflow, 788

anchors (ECMAScript), 582

angle brackets, C++11, 324

anonymous namespaces, 292

API (application programming interface), 88

- exposed interface, 124

arguments

- iterator template arguments, 551
- iterators, 532
- superclass method, 260–261
- types, operator overloading, 414
- type-safe variable-length argument lists, 692–694
 - variable-length argument lists, 310–312

arithmetic function objects, 543

arithmetic operators, 543

- increment/decrement, 420–421
- overloading, 207–209
 - shorthand operators, 208–209
 - unary minus and unary plus, 420

array class, 497–498

array container, 450

arrays, 18–20

- associative arrays, 427–428
- constant expressions, 18

deleting, 28
 dynamic memory, 27–28
 deleting, 714–715
 types, 712–713
 dynamically allocated, 713
 heap, multi-dimensional, 717–718
 of objects, 713–714
 pointers, 718–720
 stack, multi-dimensional, 715–716
 standard C-style arrays, 523
`std::array`, 20
 vector container and, 449–450

ASCII (American Standard Code for Information Interchange), 575

assert macro, 839–840
 `static_assert`, 840–841

assign() method, 476–477

assigning object value to objects, 170–173

assignment
 bitwise, 180
 disallowing, 188
 self-assignment, 186
 shallow, 180

assignment compared to copying, 173–175

assignment operator, 170
 declaring, 171
 defining, 172–173
 helper routines, 187
 initializer-lists, 664–665
 move assignment operator, 283–286
 Spreadsheet class, 185–187

associative arrays, 427–428

associative containers, 453, 468
 hashmaps as, 658–670

map
 constructing, 506
 element lookup, 509
 element remove, 509–510
 example, 510–511
 `insert()` method, 507
 iterators, 508–509
 `operator[]`, 508
 types, 505–511

method requirements, 659–661
`multimap`
 element lookup, 512
 example, 512–514
`multiset`, 516
`set`, 515–516
 typedefs, 658–659
 unordered, 516–522
 `unordered_map`, 519–521
 `unordered_multimap`, 522
 `unordered_multiset`, 522
 `unordered_set`, 522

atomic operations, 757–758
 library, 755

atomic types, 755–757

attributes, C++11, 306–307

`auditVoterRolls()` function, 571

auto keyword, 23, 402, 480, 507, 570, 602, 617, 699
 templates, 689–691

`auto_ptr`, 728

B

back references, 583, 587
`back_insert_iterator` class, 631–632
`bad()` method, 350
`bad_alloc` exception, 397–398
 base classes
 abstract, 238–239
 ambiguity, 248
 source code, 239
 virtual base, 248
 virtual base classes, 270–271

`begin()` method, 523

behaviors, 110
 overriding, 115

bibliography, 885–894

bidirectional I/O, 366–367

big-O notation, 91–93

binary logical operators, overloading, 421–422

binary operators, 11

`binary_search()` function, 566

`bind2nd()` method, 548

bitset, 524–525
 bitwise operators, 525–526
 example, 526–529
bitset container, 453–454
bitwise copying, 180
bitwise function objects, 546
bitwise operators, 525–526
 overloading, 421–422
bool, vector and, 490–491
bool() conversion operator, 373
bool variable, 10
Boolean expressions, conversion operators, 434–435
bounded repeats, 583
brackets for deleting arrays, 28
break keyword, 21
buffer overflow errors, 736
buffered streams, 346, 349
buffers, packet buffers, 499–501
bugs
 avoiding, 828–829
 catastrophic bugs, 828
 cosmetic bugs, 828
 debug traces, 831–834
 error logging, 829–831
 inheritance, 218
 noncatastrophic bugs, 828
 planning for, 829–841
 reproducing, 842–843
 debugging nonreproducible, 843–844
 debugging reproduced, 843
 root cause, 828
built-in classes, vectors, 476
business modeling workflow, 788

C

C++
 as object-oriented language, 34–36
 Stroustrup, Bjarne, 4
C++ design
 abstraction, 84–85
 differences, 82–83
 reuse, 85–87
C standard library, 97–98

C++ Standard Library, 443
compile-time rational arithmetic, 446
exceptions, 446
header files, 895–897
 algorithms, 898
 allocators, 898
 containers, 897–898
 exceptions, 899
 general utilities, 898–899
 I/O streams, 900
 iterators, 898
 mathematical utilities, 899
 threading library, 901
I/O streams, 445
localization, 445
mathematical utilities, 446–447
random numbers, 447
regular expressions, 445
smart pointers, 446
STL (standard template library),
 containers, 448–49
 strings, 445
 time utilities, 447
 tuples, 447
C++11, 303
 angle brackets, 324
 attributes, 306–307
 containers, changes, 483–484
 conversion operators, 305–306
 emplace operations, 484
 function syntax, 303
 initialization, uniform, 303–305
 initializer lists, 305
 literals, user-defined, 307–309
cache invalidation, 809
caching, design-level efficiency, 808–809
callbacks, 531
 predicate function callbacks, 533–534
calling copy constructors, explicitly, 164
calling methods, from methods, 148–149
call_once(), 763
capitalization in naming, 70
capture blocks, lambda expressions, 537
capture groups, 583
capturing, 537

cascading if statement, 16

casting

- downcasting, 235–236
- pointers, 719–720
- upcasting, 235

casting variables, 10

casts

- `const_cast`, 29
- `dynamic_cast`, 300–301
- `reinterpret_cast`, 299–300
- situations for use, 301
- `static_cast`, 298–299

catastrophic bugs, 828

catch statements, 388–389, 404

catching exceptions, 30–31, 370, 373–374

- multiple, 378–380
- stack unwinding and, 396–397

`cbegin()` method, 655–656

`cend()` method, 655–656

chaining constructors, 230

`char` exception, 377

`char` variable, 10

`char16_t` variable, 10

`char32_t` variable, 10

character classes, 584

character sets

- encodings, 577
- matches, 584–585
- non-Western, 577–579
- UCS (Universal Character Set), 77
- wide characters, 576–577
- word boundaries, 586

characters, 10

- code points, 577

chess program, 99–104

Chrono library

- clocks, 610–611
- duration, 606–609
- `time_point` class, 611–612

class templates, 317

- friend function templates, 342–343
- grid classes, 320–322
- partial class specialization, 679–683
- writing, 317–324

classes

- abstraction, 130
- access specifiers, 145–146
- array, 497–498
- `back_insert_iterator`, 631–632
- base classes
 - abstract, 238–239
 - virtual, 248
- built-in, vectors, 476
- character classes, 584
- `condition_variable`, 767–769
- data members, 34
 - access, 148
- declaring, 34–35
- definitions, 144–147
- derived, 218
- `DoubleSpreadsheetCell`, 239
- duration, 606–609
- enumerated types, 201
- exception, writing, 390–392
- exending, 218–221
- friends, 202–203
- `fstream`, 366–367
- generic template class, 133
- hierarchies, 101–102
 - diamond-shaped, 248
- `clear()` method, 668

clients, inheritance and, extending classes, 219–220

clocks, 610–611

code

- compiling, 5
- decomposing, 22
- reusable
 - advantages, 89
 - design, 128–130
 - ease of use, 139–140
 - generality, 139–140
 - writing, 86
- reusing
 - C standard library, 97–98
 - capabilities, 90–91
 - categories of available code, 87
 - disadvantages, 89–90
 - frameworks, 88

code (*continued*)
 libraries, 88
 licensing, 94
 limitations, 90–91
 open source libraries, 96–97
 performance, 91–94
 platform limitations, 94
 polymorphism and, 116
 prototype, 95
 stand-alone functions/classes, 87
 strategies, 90–95
 support, 94
 third-party application bundles, 95–96

code points, 577

coding, templates, 444–445
 coding without, 317–320

coercing variables, 10

comments, 4–5
 code reuse design, 138
 C-style, 4–5
 reasons for
 complicated code explanation, 114–115
 metainformation conveyance, 61–62
 usage explanation, 58–60

 style
 ad hoc, 65
 every line, 62–63
 fixed-format, 64–65
 prefix comments, 64
 self-documenting, 66

comparison algorithms, 459, 553–555

comparison function objects, 544–545

comparison operators, overloading, 210–211

comparisons, vectors, 477

compiler, template processing, 322

compiler-generated constructors, 168–169

compiler-generated default constructors, 158–159

compile-time debug mode, 834–835

compile-time rational arithmetic, 446

compiling code, 5

components, 109

condition variables, 767–769

conditional operators, 17–18

conditionals
 conditional operators, 17–18
 if/else statements, 16
 switch statements, 16
 ternary operator, 17

condition_variable class, 767–769

configuration management workflow, 788

const characters, string literals and, 50

const data members, 190–191

const iterator, 649

const keyword
 methods, 289
 parameters, 32, 287
 pointers, 286–289
 references, 288–289
 variables, 287

constants, 71
 const keyword, 32
 namespaced, 70

const_cast, 298

constexpr keyword, 289–290

constructor inheritance, 253–256

constructors, 35
 chaining, 230
 compiler-generated, 168–169
 copy constructors, 162–165
 calling explicitly, 164
 subclasses, 264–265

 default, 156–161
 compiler-generated, 158–159
 explicity defaulted, 159
 explicity deleted, 160

 delegating, 156

 delegating constructors, 167–168

 DoubleSpreadsheetCell constructor, 241

 error handling, 400–402

 function-try-blocks, 402–404

 hashmap class, 640–641, 663–664

 on heap, 154–155

 initializer-list constructors, 165–167

 move constructors, 283–286

multiple, 155–156
 parent constructors, 229–230
 on stack, 154
 vectors, 475–476
 writing, 153–154
 zero-initialization, 476

containers, 468–469
 adapters, 468
 queue, 498–501
 associative, 368, 504
 map, 505–511
 multimap, 512–514
 multiset, 516
 set, 515–516

bitset, 524
 bitwise operators, 525–526
 example, 526–529

C++ standard library, 443
 C++11 changes, 483–484
`forward_list`, forward iterators, 450
 hash tables, 468
 header files, 898
 interface, 449
 methods, 645–646
 reversible, 658
 sequential, 468
 array class, 497–498
 deque, 491
`forward_list`, 495–497
 hashmap, 670–671
 list, 492–495
 vector, 473–490

standard C-style arrays, 524

STL (standard template library), 448–449
 array, 450
 bitset, 453–454
 deque, 50
`forward_list`, 450
 hash tables, 453
 list container, 340
 map, 452–453
 multimap, 452–453
 multiset, 452
 priority_queue, 451
 set, 452

stack, 451–452
 table listing, 454–456
 unordered associative containers, 453
 vector, 449–450

streams, 524
 strings, 524
 typedefs, 648
 vector, 280–281
 writing, hashmap, 636–644

`continue` keyword, 21

conversion
 implicit, 205–206
 numeric, 53–54
 operators
 C++, 11

conversion operators
 ambiguity, 433–434
`bool()`, 373
 Boolean expressions, 434–435
 writing, 432–436

`copy()` function, 558–559

copy constructors, 162–165
 assignment comparison, 173–175
 calling explicitly, 164
 ctor-initializers, 163
 helper routines, 187
 object members and, 174–175
 Spreadsheet class, declaration, 185
 subclasses, 264–265

`copy_backward()` function, 558

copying
 bitwise, 180
 shallow, 180

`copyString()` function, 48–49

core process workflows, 788

cosmetic bugs, 828

counters
 naming conventions, 69
 static data members, 189

`count_if()` algorithm, 542, 602

`cout` stream, 348

covariant return types, 248

CPUs, 742

cruft, 67

`c_str()` method, 52–53

C-style arrays, 524
C-style comments, 4–5
C-style strings, 30, 47
ctor-initializers, 144, 160–162
 copy constructors, 163
 data member initialization, 160–161
curly braces, alignment and, 72–73

D

dangling pointers, 182, 446, 739
data members, 34
 access, 148
 ambiguity, 247
 const, 190–191
 const reference data members, 192–193
 const reference members, 190–191
 mutable, 95
 reference data members, 191–192
 reference members, 191–192
 references, 276
 static, 188–190, 291
 accessing, 189–190
 methods, 189–190
d-char-sequence, 55
deadlocks, 745–746
deallocate() method, 628
deallocation operators, 436–442
debug mode
 compile-time, 834–835
 ring buffers, 835–839
 run-time, 835
 start-time, 831–834
debugging
 article citation example, 850–861
 debug traces, debug mode,
 831–834
 Fundamental Law of Debugging, 828
 introduction, 827
 Linux, GDB debugger, 857–859
 memory
 class-related errors, 848
 error categories, 847–848
 general errors, 848–849
 object errors, 848

multithreaded programs, 849–850
reproducing bugs, 842–843
 debugging nonreproducible,
 843–844
 debugging reproduced, 843
 symbolic debugger, 843
Visual C++ 2013 debugger, 859–861
declarations, 5
 assignment operators, 171
 classes, 34–35
 forward declaration, 191
 functions, 22
 methods, order, 146–147
 pointers, 26
declaring variables, 9
decltype keyword, 689
decltype keyword, templates, 689–691
decltype variable, 10
decomposing code, 22
decomposition, 66–68
 modular, 67
 refactoring, 67
decrement() method, 650–651
decrement operator (--), 12
decrement operators, overloading, 420–421
deduction rules, 684
default constructors, 156–161
 compiler-generated, 158–159
 explicitly defaulted, 165
 explicitly deleted, 160
default parameters, 197
#define directive, 5
definitions, 5
 classes, 144–147
delegating constructors, 156, 167–168
delete keyword, 709–710
delete-expression, 436
demote() method, 39
deployment workflow, 788
deque container, 450, 491
dereferencing operators, 429–432
dereferencing pointers, 26, 719
derived classes, 218
design. *See also* C++ design; programming
 design; software design

- abstraction and, 84–85
- chess program, 99–104
- reusable code, 128–129
 - abstraction, 129–130
 - aggregation, 132
 - checks and safeguards, 134
 - class hierarchies, 131
 - comments, 138
 - documentation, 138
 - interfaces, 134–139
 - separate concepts, 130–132
 - subsystems, 130–131
 - templates, 132–134
 - user interface dependencies, 132
- threading, 777–778
- design patterns, 98
- design techniques, 98
 - classes, template, writing, 972–973
 - double dispatch, 973–978
 - throwing/catching exceptions, 370
- design-level efficiency, 802
 - caching, 808–809
 - object pools, 809–813
- destination ranges, 556
- destructors, 169–170
 - error handling and, 404–405
 - freeing memory, 180
 - `ifstream`, 362
 - `ofstream`, 362
 - vectors, 475–476
 - virtual, 232
 - method templates, 330
- diamond-shaped hierarchies, 248
- directives
 - preprocessor, 5–6
 - using, 8
- distributions, random numbers, 618–621
- division operator (/), 12
- DLL (Dynamic Link Libraries), 296
- documentation
 - code reuse design, 138
 - comments
 - reasons for, 58–60
 - styles, 62–66
- double deletion, 739
 - smart pointers, 732–733
- double precision numbers, 10
- double variable, 10
- double-checked locking algorithm, 766–767
- `doubleDelete()` function, 732–733
- double-ended queue. *See deque container*
- double-freeing, 446
- `DoubleSpreadsheetCell` class, 239
 - constructor, 241
- `DoubleSpreadsheetCell` constructor, 241
- `doubleToString()` method, 193
- doubly linked list structures, 450
- `do/while` loop, 21
- downcasting, 235–236
- `doWorkInThread()` function, 753
- dumb pointers, memory allocation results, 739
- `duration` class, 606–609
- dynamic arrays *versus* dynamically allocated arrays, 713
- dynamic memory, 707
 - advantages, 708
 - allocation, 177–188
 - `delete` keyword, 709–710
 - failure, 711
 - `malloc()` function, 710–711
 - `new` keyword, 709–710
 - arrays, 27–28
 - deleting, 714–715
 - multi-dimensional, 715–718
 - types, 712–713
 - freeing, 180
 - mental model, 708–709
 - pointers, 25–29
- dynamic strings
 - C-style strings, 48–50
 - raw string literals, 54–55
 - `string` class, 51–53
 - string literals, 50
- dynamically allocated arrays, 713
- `dynamic_cast`, 300–301
- dynamic-length vectors, 475

E

ease of use, 139–140
ECMAScript, 582
 alternation, 582
 anchors, 582
 back references, 587
 character sets, matches, 584–585
 grouping, 583
 precedence, 584
 raw string literals, 587–588
 regular expressions, 587–588
 repetition, 583
 wildcards, 582
 word boundaries, 586
efficiency, 801–802
 design-level, 802, 808
 caching, 808–809
 object pools, 809–813
 language-level, 802, 803–808
 catching exceptions by reference, 806
 move semantics, 806
 pass-by-reference, 804–805
 return by reference, 805
 temporary objects, 806–807
elements
 list container, 492–495
 map associative container, 509–510
 multimap associative container, 512
 objects, fields, 479
 requirements, 468–469
 value semantics, 468–469
vectors
 access methods, 474–475
 appending to, 481–485
 initial value, 474–475
 user-defined classes, 476
`emplace()` method, 499
emplace operations, 484–485
 `hashmap` class, 666–667
employee records system sample program
 Database class
 `Database.cpp` file, 42–43
 `Database.h` file, 41–42
 `DatabaseTest.cpp` file, 44

Employee class
 `Employee.cpp` file, 39–40
 `Employee.h` file, 38–39
 `EmployeeTest.cpp` file, 41
user interface, `UserInterface.cpp` file, 44–46
`empty()` function, 548
emulation, function partial specialization, 683–684
`enable_if`, 703–705
`end()` method, 524
`#endif` directive, 6
`endl` manipulator, 351
end-of-file, 357
end-of-line sequence, 6
enumerated types, 13–14
 in classes, 201
 strongly typed enumerations, 14–15
 type-safe, 14–15
`eof()` method, 357–358
`equal()` function, 553–555
`equal_range()` function, 566
`erase()` function, 561–562
`erase()` method, 514
error categories, 847–848
error checking, 470
error handling, 104
 constructors, 400–402
 function-try-blocks, 402–404
 destructors, 404–405
 exceptions, 369–370
 input, 357–358
 memory allocation errors, 397–400
 new handler callback function, 399
 output, 350
error level, debugging, 831
error logging, 829–831
`ErrorCorrelator`, 503
escape character, n, 7
event objects, 767
exception classes, writing, 390–392
exception handling, 376
 `unexpected_handler`, 384
 unhandled exceptions, 380–382

exception_ptr, 752–753
exceptions, 30–31, 387–389, 470
 bad_alloc, 397–398
 benefits, 371–372
 C++, 371–372
 C++ standard library, 446
 catching, 370, 373–374
 multiple, 378–380
 by reference, 806
 stack unwinding and, 396–397
 char, 377
 class hierarchy and, 387–389
 custom, 72
 header files, 899
 hierarchy, 387–388
 introduction, 369
 invalid_argument, 374
 lambda expressions, 536
 matching, 380
 multithreaded programming, 752–754
 nested, 392–394
 nested_exception mixin class, 392–394
 polymorphism and, 387–388
 runtime_error, 380
 specifications, 382–387
 throw lists, 382–387
 throwing, 370, 373–374
 multiple, 378–380
 stack unwinding and, 396–397
 types, 376–378
 unexpected, 383
 unhandled, 380–382
 execution order of operators, 13
 explicitly calling copy constructors, 164
 explicitly defaulted constructors, 159
 exposed interface, 123–125
 API (application programming interface), 124
 audience considerations, 123
 subsystem interface, 124
 utility classes *versus* libraries, 124
 expressions
 regular expressions, 445, 581–582
 short-circuit logic, 18

extending, 114
 classes, inheritance and, 219–220
 extending classes, 218–221
 extern keyword, 292–293
 external linkage, 291, 291–292
 extraction operators, overloading, 422–423

F

facets, 580–581
 factorials at compile time, 696
 fail() method, 350
 feature creep, 782
 file streams, 362–363
 ifstream destructor, 362
 ofstream destructor, 362
 seek() method, 363–365
 tell() method, 363–365
 files
 header files, 309–310
 objects, linking, 5
 FILO (first-in, last-out), 451
 final keyword, 221
 find() algorithm, 634–635
 find() function, 532–535, 551–553
 reverse_iterator, 629
 Find() function, template, 339–340
 find_all() algorithm, 634–635
 findElement() method, 641
 find_if() function, 532–535, 546
 fixed-format comments, 64–65
 fixed-length arrays, vectors, 474–475
 fixed-length vectors, 474–475
 float variable, 10
 floating-point numbers, 10
 flush() method, 349, 366
 flush-on-access, 365–366
 for loop, 21
 range-based, 21–22, 478–479
 for_each() function, 562, 562–564
 formatting
 alignment, curly braces, 72–73
 parentheses, 74
 spaces, 74
 tabs, 74

forward declaration, 191
`forward_list` container, 450, 495–497
 forward iterators, 450
frameworks
 code reuse, 88
free software, 96
freeing memory, destructors and, 180
friend function templates to class templates, 342–342
friend keyword, 202–203
friends, 202–203
`fstream` class, 366–367
`func()` method, 386
function adapters, binders
 `bind2nd()`, 548
 `std::bind()`, 546–548
function objects, 428, 531, 542
 adapters, binders, 546–548
 adapting functions, 549
 arithmetic function objects, 543
 bitwise function objects, 546
 comparison function objects, 544–545
 logical function objects, 545
 member functions, 548–549
 negators, 548
 threads, 748–750
 writing, 549–550
function pointers, 725–726
 implementing, 602–603
 threads, 746–748
 typedefs, 296–297
function templates, 339–340
 overloading, 341–342
 specialization, 340–341
functional relationships, 119–120
functionality
 adding, 115
 replacing, 115
 required, user interface, 136–137
function-call operator, 428–429
functions, 22–23, 316. *See also* algorithms
 `accumulate`, 570
 `addOne()`, 30
 `allTrue()`, 545
 alternative function syntax, 338, 338–339
 `auditVoterRolls()`, 571
 `binary_search()`, 566
 C++11 syntax, 303
 callbacks, predicate function callbacks, 533–534
 `copy_backward()`, 558
 `copyString()`, 48–49
 as data, 725–726
 declarations, 22
 `doubleDelete()`, 732–733
 `doWorkInThread()`, 753
 `empty()`, 548
 `equal()`, 553–555
 `equal_range()`, 566
 `Find()`, 339–340
 `find()`, 532–535, 634–635
 `find_all()`, 634–635
 `find_if()`, 532–535
 `for_each()`, 562–564
 friends, 202–203
 `funcTwo()`, 395
 `getDuplicates()`, 572–573
 hash functions, 517–518
 `includes()`, 566–569
 `incr()`, 281
 `lexicographical_compare()`, 553–555
 `lower_bound()`, 566
 `main()`, 6
 `make_move_iterator()`, 633
 `malloc()`, 710–711
 `malloc_int()`, 730
 `max()`, 482
 `mem_fn()`, 548–549
 `min()`, 482
 `minmax()`, 482
 `mismatch()`, 553–555
 namespaces and, 7–8
 `operator()`, 428–429
 parameters, 22
 partial specialization, overloading and, 683–684
 `printVector()`, 482
 procedures and, 108

process(), 602–603
processValues(), 692
prototypes, 22
random_shuffle(), 460
readName(), 354
realloc(), 713
sizeof(), 49
sort(), 565–566
stand-alone, reuse and, 87
static, 292
strcat(), 49
strcpy(), 48
strlen(), 48–49
swap(), 482, 647
testCallback(), 541
threadFunc(), 753
throw_with_nested(), 393
unexpected(), 384
upper_bound(), 566
function-try-blocks, 402–404
functors, 428
funcTwo() function, 395
Fundamental Law of Debugging, 828
futures, 770–772

G

garbage collection, 724
GDB debugger, 857–859
generality, 139–140
generate() algorithm, 542
generating random numbers, 612–621
 random number engines, 613–615
 predefined, 616
generic programming, 444
 reusable code and, 316
 templates and, 316
generic template class, 133
get() method, 354–355
getDuplicates() function, 572–573
getline() method, 357
getSize() method, 359
getString() method, 240
 renaming, 195
getters/setters, 70

getValue() method, 145
 renaming, 195
global functions, operators, 413–414
GNU project, 96
good() method, 350, 357–358
GPL (GNU Public License), 96
gprof, 814–822
graphics cards, cores, 742–743
greedy repeats, 583
grouping (ECMAScript), 583

H

handles, 129
has-a relationship, 113–114
 is-a relationship comparison, 116–119
hash functions, 517–518, 636, 636–637
hash tables, 68, 453, 516–522, 636. *See also*
 unordered associative containers
 unordered_map, 519–521
hash_map, *implementation*, 637
hashmap class
 accessor operations, 668–670
 as associative container, 658–670
 clear operation, 668
 constructor, 640–641, 663–664
 emplace operations, 666–667
 erase operations, 667–668
 implementation, 640–643
 initializer list assignment operator, 664–665
 initializer list constructor, 664
 insertion operations, 665
 interface, 639–640
 sequential containers, 670–671
 as STL container, 644–657
 template demonstration, 643–644
 templates, 637–638
 typedefs, 648–649
header files, 309–310
 algorithms, 898
 allocators, 898
 containers, 897–898
 exceptions, 899
 general utilities, 898–899
 iterators, 898

header files (*continued*)

- I/O streams, 900
- mathematical utilities, 899
- template definitions, 325–326
- threading library, 901
- heap, 25
 - constructors on, 154–155
 - objects on, 151–152
 - vector allocation, 476
- heap algorithms, 463
- heap arrays, multi-dimensional, 717–718
- Hello, World
 - arrays, 18–20
 - `std::array`, 20
 - comments, 4–5
 - conditionals
 - conditional operators, 17–18
 - `if/else` statements, 16
 - `switch` statements, 16–17
 - ternary operator, 17
 - functions, 22–23
 - I/O streams, 6–7
 - loops
 - `for`, 21
 - `do/while`, 21
 - `while`, 20–21
 - `main()` function, 6
 - namespaces, 7–8
 - operators, 11–13
 - preprocessor directives, 5–6
 - types
 - enumerated, 13–14
 - structs, 15
 - variables, 9–10
- helper methods, 149
- helper routines
 - assignment operator, 187
 - copy constructors, 187
- hierarchies, 120–121
 - classes, 101–102
 - exceptions, 387–389
 - diamond shaped, 242
 - exceptions, 387–388
 - parallel, 249
 - reusable code and, 131

homogenous elements, 447

Hungarian Notation, 70–71

I

- `#ifdef` directive, 6
- `if/else` statements, 16
 - cascading `if` statement, 16
- `#ifndef` directive, 6
- `ifstream` destructor, 362
- `ifstream` object, `bool()` conversion operator, 373
- implementation, 212–215
 - implementation class, 212–215
 - implementation workflow, 788
 - versus* interface, 1213
- implicit conversion, 205–206
- In-class member initializers, 167
- `#include` directive, 5
- includes, preprocessor directives, 6
- `includes()` function, 566–569
- `incr()` function, 281
- `increment()` method, 650–651
- increment operator `(++)`, 12
 - overloading, 420–421
- indexing, iterators and, 481
- inheritance, 114, 217–218
 - bug fixes and, 218
 - clients and, extending classes, 219–220
- constructors, 253–256
- derived classes, 218
- functionality
 - adding, 115
 - replacing, 115
- multiple, 121–122, 244–245
 - uses, 248–249
- non-public, 270
- parent classes, 218
- pointers, 225
- polymorphism, spreadsheet, 236–237
- preventing, 221
- properties, 115
 - replacing, 115

reuse
 subclasses and, 226–228
`WeatherPrediction` class, 225–226

specialization comparison, 337
 subclasses, 218–219
 superclasses, 218
 templates comparison, 133–134

initialization
 constructors, 161–163 (*See also* ctor-initializers)
 in-class member initializers, 167
 lists (C++11), 305
 nonlocal variables, 294
 references, 274
 uniform in C++, 303, 303–305, 476
 zero-initialization, template types, 328–333

`initializer_list`, 476

initializer-list constructors, 165–167
 assignment operators, 664–665
`hashmap` class, 664

`inline` keyword, 197–198

inline methods, 197–198

in-memory streams, 360–361

input
 error handling, 357–358
 locale awareness, 359
 manipulators, 358–359
 objects, 359–360
 streams, 353–354
`get()` method, 354–355
`getline()`, 357
`peek()`, 356–357
`putback()`, 356
`readName()` function, 354
`unget()`, 355–356

input stream iterator, 630

`inRange()` method, 178

`insert()` method, `map` associative container, 507

`insert` iterators, 631–632

insertion operators, overloading, 422–423

`insert_iterator` class, 631, 631–632

instantiation, 133, 321
 selective, 325

int variable, 9

integer non-type parameters, 327–329

integers, 9
 enumerated types, 13–14

interface, 211, 212
 algorithms, 449
 API (Application Programming Interface), 124
 containers, 449
 exposed, 123–125
 audience considerations, 123
 subsystem interface, 124
 utility classes *versus* libraries, 124
versus implementation, 123
 implementation class, 212–215
 interface class, 212–215
 usability, 134–139

interface class, 212–215

internal linkage, 291–292

interviews, 863–883

`intVector`, 476

`invalid_argument` exception, 374

invalidation of iterators, 485

I/O
 bidirectional, 366–367
 overview, 345–346
 streams, 6–7, 445
 header files, 900
 user input, 7

`ios_base::out`, 362

`iostream` subclass, 366–367

is-a relationship, 114–116
 has-a relationship comparison, 116–119
 polymorphism, 116

iterative processes, 785–786

iterator adapters (STL)
 insert iterators, 631–632
 move iterators, 632–633
 reverse iterators, 629–630
 stream iterators, 630–631

iterators, 457, 470–471
 arguments, 532
 categories, 471
`const`, 649
`generate()` algorithm, 542

iterators (*continued*)
 header files, 898
 indexing and, 481
 input stream iterator, 630
 invalidation, 485
 list container, 492
 map associative container, 508–509
 methods, 472–473
 operations, 481
 output stream iterator, 630
 regular expressions, 588
 safety, 480–481
 template arguments, 551
 traits, 635–636
 typedefs, 472–473, 655–656
 types, 551
 vector, 478–481
 writing, 647–648

`iterator_traits` class, 635–636

K

keywords
`auto`, 23, 339, 402, 570, 602, 689–691, 699
`break`, 21
`const`, 32, 191, 286–289
 methods, 289
`parameters`, 32, 287
`pointers`, 286–289
`references`, 288–289
`variables`, 287
`constexpr`, 289–290
`continue`, 21
`decltype`, 689–691
`delete`, 709–710
`extern`, 292–293
`final`, 221
`friend`, 202–203
`inline`, 197–198
`new`, 709–710
`noexcept`, 382–384
`override`, 252–253
`static`

functions, 293–294
 variables, 293–294
`throw`, 374
`typename`, 641
`using`, 253
`virtual`, 222

L

lambda expressions
 capture block, 537
 as parameters, 541
 return types, 536–537
 syntax, 536–539
 threading, 750
 language-level efficiency, 802, 803–808
 catching exceptions by reference, 806
 move semantics, 806
 pass-by-reference, 804–805
 return by reference, 805
 temporary objects, 806–807
`lexicographical_compare()` function, 553–555
 libraries. *See also* C++ standard library
 atomic operations, 755
 C standard library, 97–98
 Chrono
 clocks, 610–611
 duration, 606–609
`time_point` class, 611–612
 code reuse, 88
 library handles, 296
 open source, 96–97
 finding, 96–97
 using, 96–97
 Ratio, 603–606
`regex`, 588–589
`regex_iterator()`, 593–594
`regex_match()`, 589–591
`regex_replace()`, 596–599
`regex_search()`, 592–593
`regex_token_iterator()`, 594–596
 regular expressions library, 75
 standard library, 36
 classes, 36

- life cycles of objects, 152
 - assigning to objects, 170–173
 - creation, 153–169
 - destruction, 169–170
- LIFO (last-in, first-out), 451
- `linear_congruential_engine`, 615
- linkage
 - doubly linked list structures, 350
 - external, 291, 291–292
 - internal, 291–292
 - static, 291
 - static keyword, 291–292
- linking
 - object files, 5
 - streams, 365–366
- Linux, GDB debugger, 857–859
- list container, 450
 - `forward_list` comparison, 495–497
- list container, 492–495
 - elements
 - accessing, 492
 - adding/removing, 492
 - enrollment, 494–495
 - iterators, 492
 - `merge()` method, 494
 - `remove()` method, 494
 - `remove_if()` method, 494
 - `reverse()` method, 494
 - `size()`, 492
 - `sort()` method, 494
 - special operations, 493–495
 - `unique()` method, 494
- literals, 11
 - literal pooling, 50
 - string literals, 50
 - localizing, 576
 - raw string literals, 54–55, 587–588
 - user-defined, C++11, 307–309
- locales, 579
 - classic, 580
 - input and, 359
 - Windows, 579
- localization, 445, 575–576
 - string literals, 576
- lock classes, 761–763
- locks
 - double-checked locking algorithm, 766–767
 - multithreading, 761–763
- `log()` method, 831–834
- log level
 - debugging, 831
 - `Logger::log()` method, 832
- logical function objects, 545
- long double variable, 10
- long long variable, 9
- long variable, 9
- `longjmp()`, 371
- loops
 - `do/while` loop, 21
 - `for` loop, 21
 - range-based, 21–22
 - loop unrolling, 696–697
 - `while` loop, 20–21
- `lower_bound()` function, 566

M

- macros, preprocessor macros, 312–313
- `main()` function, 6
- `make_move_iterator()` function, 633
- `make_pair()` template, 505
- `make_shared()` method, 731
- `malloc()` function, 710–711
- `malloc_int()` function, 730
- map associative container
 - constructing, 506
 - elements
 - `insert()` method, 507
 - lookup, 509
 - removing, 509–510
 - example, 510–511
 - iterators, 508–509
 - `operator[]`, 508
 - types, 506
 - `unordered_map` comparison, 519–520
- map container, 452
- mark and sweep garbage collection, 775

mathematical utilities, 446–447
 header files, 899
Matrix class, 400–401
max() function, 482
member functions, 548–549
 threads, 751
members, 145
 pointers to, 727
mem_fn() function, 548–549
memory
 aliasing, 728
 allocation
 contiguous pieces, 712
 dumb pointers, 729
 failure behavior, 399–400
 null pointers, 398
 vectors, 485–486
 allocation errors, 397–400
 class-related errors, 848
 double deletion, 739
 dynamic, 707
 advantages, 708
 allocation, 177–188, 709–711
 arrays, 713
 mental model, 708–709
 multi-dimensional arrays, 715–718
 pointers, 25–29
 errors, categories, 847–848
 freeing, destructors and, 180
 function pointers, 725–726
 garbage collection, 724
 general errors, 848–849
 heap, 25
 object errors, 848
 object pools, 725
 orphaned, 184
 out-of-bounds access, 735–736
 pointer arithmetic, 722–723
 pointers, 718
 smart pointers, 727–734
 stack, 25
 strings, underallocating, 734–735
memory access errors, 847–848
memory freeing errors, 845–846
memory leaks, 446, 736–739
 Linux, 738–739
 Valgrind, 738–739
 Windows, 737–738
memory management, custom, 723–724
merge() method, 494
Mersenne twister, 615
metaprogramming. *See* template
metaprogramming
method calls, 136
method templates, 330–334
 non-type parameters, 332–334
methods, 145
 add(), overloading, 203–204
 allocate(), 628
 assign(), 476–477
 associative containers, 659–661
 bad(), 350
 begin(), 523
 bind2nd(), 548
 calling from methods, 148–149
 cbegin(), 655–656
 cend(), 655–656
 clear(), 668
 const, 194–195
 const keyword, 289
 constructors, 35
 containers, 645–646
 c_str(), 52–53
 deallocate(), 628
 declarations, ordering, 146–147
 decrement(), 650–651
 defining, 147–150
 demote(), 39
 doubleToString(), 193
 emplace(), 499
 end(), 523
 eof(), 358
 erase(), 514
 fail(), 350
 findElement(), 641
 flush(), 349, 366
 func(), 386
 get(), 354–355

getSize(), 359
getString(), 195, 240
getValue(), 145, 195
good(), 350, 358
helper methods, 149
increment(), 650–651
inline, 197–198
inRange(), 178
insert(), 507
iterators, 472–473
log(), 831–834
Logger::log(), 832
make_shared(), 731
multi, 973
namespaces and, 7–8
output, raw output methods, 349
overloaded
 add(), 203–204
 deleting, 196
 operator+ method, 204–205
overloading, 195–197
overriding, 222–225
 changing parameters, 250–252
 changing return type, 249–250
 clients and, 223–225
 overloaded superclass method, 258–259
 preventing, 225
 private superclass method, 259–260
 static methods, 257–258
 superclass method access level, 261–263
 superclass method arguments, 260–261
 throw list, 385–386
 virtual, 265–266
pointers to, 727
promote(), 39
pure virtual methods, 238–239
push(), 4499
push_back(), 475
put(), 349
rbegin(), 629, 658
refcall(), 278–279
rend(), 629, 658
set(), 196
setValue(), 145
static, 193, 291
stringtoDouble(), 193
tie(), 365–366
virtual, 222
 pure, 238–239
wait(), 768
what(), 340, 350
write(), 349
MFC (Microsoft Foundation Classes), 88
min() function, 482
minmax() function, 482
mismatch() function, 553–555
mixin, 122
 variadic templates, 694–695
mixin classes, 122
 variadic templates, 694–695
mod operator (%), 12
modifying algorithms, 459–460, 556–557
 copy(), 558
 erase(), 561–562
 iota(), 570
 move(), 559–560
 replace(), 560–561
 reverse(), 562
 transform(), 556–557
 unique(), 562
modular decomposition, 67
move() function, 559–560
move assignment operator, 283–286, 419
move constructor, 283–286
move iterators, 632
move semantics, 283–286
 algorithms, 536
 language-level efficiency and, 806
 move iterators, 632–633
MoveableClass instances, 633
move_backward() function, 559–560
moving ownership, 283

multi-dimensional arrays, dynamic memory, 715–718

multiline comments, 4

`multimap` container, 452

- element lookup, 512
- example, 512–514

multiple inheritance, 121–122, 244–245

- uses, 248–249

multiplication operator (`*`), 12

`multiset` associative container, 516

multithreaded programming, 741

- cancelling threads, 752
- condition variables, 767–769
- deadlocks, 745–746
- debugging, 849–850
- exceptions, 752–754
- futures, 770–772
- introduction, 742–743
- lambda expression, 750
- locks, 761–763
- logger class example, 772–776
- member functions, 751
- mutual exclusion, mutex classes, 759–761
- race conditions, 743–745
- results retrieval, 752
- `std::call_once()`, 763–764
- thread local storage, 751
- thread pools, 776–777
- threading design, 777–778
- threads
 - function object, 748–750
 - function pointers, 746–748

multithreading

- threading models, 101

mutability, lambda expressions, 537

mutable data members, 195

mutex classes, 759–761

- timed, 765–766
- writing to streams, 764–765

mutual exclusion, mutex classes, 759–761

MVC (Model-View-Controller), 99

- object-oriented frameworks, 132

`MyEnum`, 14–15

N

namespaces, 7–8

- anonymous, 292
- functions, 7–8
- methods, 7–8
- namespaced constants, 70

naming

- ambiguity, 246–247
- base classes, 248

conventions

- capitalization, 70
- counters, 69
- Hungarian Notation, 70–71
- namespaced constants, 70
- prefixes, 69
- getters, 70
- selecting, 68–69
- setters, 70

narrowing, uniform initialization, 304

N-dimensional grids, template recursion and, 686–689

negators, 548

nested classes, 199–201

nested exceptions, 392–394

`nested_exception` mixin class, 392–394

`new` keyword, 709–710

`new-expression`, 436

`NGrid` template class, 686–689

`noexcept` keyword, 382–384

noncatastrophic bugs, 828

non-integral array indices, 427–428

nonlocal variables, order of

- initialization, 294

non-modifying algorithms, search algorithms, 551–553

non-public inheritance, 270

nonstandard strings, 30

non-type template parameters, 327–329, 678–679

- method templates, 332–334
- pointers, 679
- reference, 679

non-Western character sets, 577–579

`nothrow`, 398

null pointers
 dereferencing, 26
 memory allocation routines, 398
 numbers, rational, 604
 numeric conversion, 53–54
 numerical processing algorithms,
 464, 570

O

object pools, 725
 design-level efficiency, 809–813
 object-oriented languages
 C++ as, 34–36
 philosophy
 behaviors, 110
 classes, 108–109
 components, 109
 properties, 109
 objects, 111–113, 316
 aggregation, 113–114
 arrays of, 713–714
 constructors, writing, 153–154
 elements, fields, 479
 even objects, 767
 files, linking, 5
 function objects, 448, 531
 adapters, 542–550
 adapting functions, 549
 arithmetic function objects, 543
 comparison function objects,
 544–545
 logical function objects, 545
 member functions, 548–549
 negators, 548
 writing, 549–550
 on heap, 151–152
 inheritance, multiple, 121–122
 input, 359–360
 life cycles, 152
 assigning to objects, 170–173
 creation, 153–169
 destruction, 169–170
 memory, dynamic allocation, 177–188
 output, 359–360

passing by reference, 164–165
 relationships
 has-a, 113–114, 116–119
 hierarchies, 120–121
 is-a, 114–119
 mixin classes, 122
 not-a, 119–120
 as return values, 173–174
 on stack, 151
 destruction, 169
 out of scope, 169
 temporary, 806–807
`ofstream` destructor, 362
 OOP (object-oriented programming), 34
 opaque classes, 129
 open source libraries, 96–97
 finding, 96–97
 using, 96–97
 operational algorithms, 461, 562–564
`operator*`, 430–431
`operator->`, 431–432
`operator->*`, 432
`operator[]`, 426–427
 map associative container, 508
`operator()` function, 428–429
`operator delete`, 437
 overloading, 437–439
`operator+` method, 204–205
 global, 206–207
`operator new`, 437
 overloading, 437–439
 operator overloading, 136, 155–156, 444
 415, 649
`add()` method, 203–204
 allocation/deallocation operators,
 436–442
 argument types, 414
 arithmetic, 207–209
 increment/decrement, 420–421
 shorthand operators, 208–209
 unary minus and unary plus, 420
 behavior choices, 415
 binary logical operators, 421–422
 bitwise operators, 421–422
 comparison, 210–211

operator overloading (*continued*)
dereferencing operators, 429–432
extraction operators, 422–423
function-call operator, 428–429
implementing overloaded, 243
insertion operators, 422–423
limitations, 412–413
methods *versus* global functions, 413–414
operator `delete`, 437–440
operator+ method, 204–205
 global, 206–207
overloadable operators, 415–418
placement new, 437
reasons for, 412
return types, 414–415
subscripting operator, 423–428
 non-integral array indices, 427–428
 read-only access, 426–427
type building, 211
use, 136

operators

- `%=`, 12
- `&`, 12
- `&=`, 12
- `*=`, 12
- `+=`, 12
- `-=`, 12
- `/=`, 12
- `<<`, 12
- `<<=`, 12
- `>>`, 12
- `>>=`, 12
- `^`, 12
- `^=`, 12
- `|`, 12
- `|=`, 12

arithmetic, 543
arity, 412
assignment operator, 170
 declaring, 171
 defining, 172–173
 move assignment operator, 283–286
Spreadsheet class, 185–187

associativity, 413
binary, 11
bitwise, 525–526
conditional, 17–18
conversion
 ambiguity, 433–434
 `bool()`, 373
 Boolean expressions, 434–435
 C++11, 305–306
 writing, 432–436
dereferencing, 429–432
execution order, 13
function-call, 428–429
global functions, 413–414
overloading, 359–360
placement new, 437
precedence, 413
relational, 419–420
scope resolution, 147
sequencing operator, 415
string class, 51–53
ternary, 11, 17
unary, 11

orphaned memory, 184
`ostream_iterator` class, 630
out scope (objects on stack), 169

output
 error handling, 350
 manipulators, 351–352
 objects, 359–360
 overview, 347–348
 raw output methods, 349
 standard output, 348
 streams, 347–352
 buffered, 349
 `flush()` method, 349
 `put()` method, 349
 `write()` method, 349

output stream iterator, 630
overload resolution, 196
overloading
 function partial specialization and, 683–684
 function templates, 341–342
 methods, 195–197

- deleting overloaded, 196
- `operator+ method`, 204–205
- `operator+ method (global)`, 206–207
- operators
 - arithmetic, 207–209
 - implementing overloaded, 243
 - superclass method, 258–259
- overloading operators, 136, 155–156, 359–360
 - allocation/deallocation operators, 436–442
 - argument types, 414
 - arithmetic operators
 - increment/decrement, 420–421
 - unary minus and unary plus, 420
 - behavior choices, 415
 - binary logical operators, 421–422
 - bitwise operators, 421–422
 - comparison operators, 210–211
 - dereferencing operators, 429–432
 - extraction operators, 422–423
 - function-call operator, 428–429
 - insertion operators, 422–423
 - limitations, 412–413
 - `operator delete`, 437–439
 - operators not to overload, 415
 - overloadable operators, 415–418
 - placement new, 437
 - reasons for, 412
 - return types, 414–415
 - subscripting operator, 423–428
 - non-integral array indices, 427–428
 - read-only access, 426–427
 - type building, 211
 - use, 136
- `override keyword`, 252
- overriding
 - behaviors, 115
 - methods, 222–225
 - changing parameters, 250–252
 - changing return type, 249–250
 - clients and, 223–225
 - overloaded superclass method, 258–259
- preventing, 225
- private superclass method, 259–260
- static methods, 257–258
- superclass method access level, 261–263
- superclass method arguments, 260–261
- syntax, 223
- throw list, 385–386
- virtual, 265–266
- `what()`, 340

P

- packets, buffers, 499–501
- `pair class`, 504–505
- parallel hierarchies, 249
- parameter packs, 694
- parameters, 316
 - `Allocator`, 473
 - `const keyword`, 287
 - `const keyword and`, 32
 - `const reference parameters`, 32–33
 - `default`, 197
 - functions, 22
 - lambda expressions, 536, 541
 - overridden methods, 250–252
 - references, 276–277, 279–281
 - template template parameters, 676–678
 - templates, 316
 - integer non-type parameters, 327–329
 - method templates, 332–334
 - non-type parameters, 327–329, 678–679
 - parameter packs, 694
 - type parameters, 674–676
 - `vector`, 473
- parent classes, 218
 - parent constructors, 229–230
 - private, 270
 - protected, 270
 - references, 232–234
- parent constructors, 229–230
- parent destructors, 230–232

parenthese in code, 74
partial specialization
 classes, 679–683
 functions, 683–684
pass-by-reference, efficiency, 804–805
passing
 objects, by reference, 164–165
 by value, disallowing, 188
 variables
 by reference, 27
 by value, 27
pattern-matching, 445
patterns. *See* design patterns
 subsitution pattern, 582
peek() method, 356–357
performance, 801–802
 reuse and, 91–94
placement new operators, 437
pointer arithmetic, 722–723
pointers, 718
 arrays, 718–720
 casting and, 719–720
 const keyword, 286–289
 dangling, 182, 446, 739
 declaring, 26
 dereferencing, 26, 719
 dynamic memory, 25–29
 function pointers, 725–726
 threads, 746–748
 inheritance and, 225
 to members, 727
 mental model, 718–719
 to methods, 727
 non-type template parameters, 679
 to references, 275–276
 versus references, 71–72, 278–281
 references from, 277
 references to, 275–276
 smart pointers, 28–29, 727–734
 auto_ptr, 728
 C++ standard library, 446
 class writing, 728–734
 reference counting, 732–733
 shared_ptr, 731–732
 stack unwinding and, 396
unique_ptr, 729–731
weak_ptr, 734
structures, 27
this, 150–151
typedefs, 296–297
variables, 27
polymorphism, 116
 exceptions and, 387–388
inheritance, spreadsheet, 236–237
 SpreadsheetCell hierarchy, 241–243
POSIX, initializing semaphores, 371
#pragma directive, 6
precedence, ECMAScript, 584
precision numbers, double precision numbers, 10
predicate function callbacks, 533–534
prefix comments, 64
preprocessor directives, 5–6
 #define, 5
 #endif, 6
 #ifndef, 6
 #endif, 6
 #include, 5
 includes, 6
 #pragma, 6
preprocessor macros, 312–313
printf(), 345
printing, tuples, 698–699
printVector() function, 482
priority_queue container adapter, 501–503
 example, 502–503
 operations, 501–503
private, superclasses, 259–260
procedures, 316
 functions and, 108
process() function, 602–603
processValues() function, 692
profiling, 813–814
 gprof example, 814–822
 Visual C++ 2013 example, 822–825
programming
 generic, 444
 multithreaded (*See* multithreaded
 programming)

programming design, 80
 importance of, 80–82
 programs, employee records sample program, 37–46
`promote()` method, 39
 properties, 109
 inheritance and, 115
 replacing, 115
 protected methods, inheritance, 220
 public methods, inheritance, 220
 pure virtual methods, 238–239
`push()` method, 499
`push_back()` method, 475, 484–485
`put()` method, 349
`putback()` method, 356

Q

queue container adapter
 example, 499–501
 operations, 499
 queue container adapter, 498–501

R

race conditions, 743–745
 random number engine adapter, 447, 615–616
 random number engines, 447, 613–615
 `linear_congruential_engine`, 615
 Mersenne twister, 615
 predefined, 616
 `subtract_with_carry_engine`, 615
 random numbers
 C++ standard library, 447
 distribution, 618–621
 generating, 612–621
`random_device` engine, 614–615
`random_shuffle()` function, 460
 range specification, 584
 range-based `for` loop, 21–22
 ranges
 destination, 556–557
 source, 556

Ratio library, 603–606
 rational numbers, 604
 raw output methods, 349
 raw string literals, 54–55
`rbegin()` method, 629, 658
`r-char-sequence`, 55
`readName()` function, 354
 read-only access, 426–427
`realloc()` function, 713
 recursion, templates, 685–689
 refactoring, decomposition and, 67
`refcall()` method, 278–279
 reference counting smart pointers, 732–733
 reference data members, 191–192
 `const` reference data members, 192–193
 reference parameters, 279–281
 `const`, 32
 references, 29–30
 back references, 583, 587
 `const` keyword, 288–289
 data members, 276
 initializing, 274
 modifying, 275
 non-type template parameters, 679
 parameters, 276–277
 parent classes, 232–234
 passing objects by, 164–165
 passing variables by, 27, 277
 from pointers, 277
 to pointers, 275–276
 versus pointers, 71–72, 278–281
 pointers to, 275–276
 reference variables, 274–276
 return values, 278
 rvalue, 281–286
 unnamed values, 274
 regex library, 588–589
 `regex_iterator`, 593–594
 `regex_iterator()`, 593–594
 `regex_match()`, 589–591
 `regex_replace()`, 596–599
 `regex_search()`, 592–593
 `regex_token_iterator()`, 594–596
 typedefs, 588–589

regular expressions, 445, 581–582
 ECMAScript, 582–588
 iterators, 588
 raw string literals, 587–588
 regex library, 588–589
regular expressions library, 575
`reinterpret_cast`, 299–300
relational operators, 419–420
relationships
 functional, 119–120
 has-a, 113–114, 116–119
 hierarchies, 120–121
 inheritance, multiple, 121–122
 is-a, 114–116, 116=119
 mixin classes, 122
 not-a, 119–120
`remove()` method, 494
`remove-erase-idiom`, 482, 561–562
`remove_if()` method, 494
`rend()` method, 629, 658
repeats, greedy, 583
repetition (ECMAScript), 583
`replace()` function, 560–561
replace operations, 583
required functionality, user interface, 136–137
requirements on elements, 468–469
requirements workflow, 788
return by reference, 805
return types
 lambda expressions, 536–537
 operator overloading, 414–415
return values
 objects as, 173–174
 references, 278
returning values, 22–23
reusable code
 abstraction, 129–130
 aggregation, 132
 checks and safeguards, 134
 class hierarchies, 131
 comments, 138
 design, 128–129
 documentation, 138
 ease of use, 139–140
 generality, 139–140
 generic programming, 316
 interfaces, usability, 134–139
 separate concepts, 130–132
 subsystems, 130–131
 templates, 132–134
 user interface dependencies, 132
reusable code and, 131
reuse
 advantages, 88–89
 C standard library, 97–98
 capabilities, 90–91
 categories of available code, 87
 disadvantages, 89–90
 frameworks, 88
 ideas, 87
 inheritance and
 subclasses and, 226–228
 `WeatherPrediction` class, 225–226
 libraries, 88
 open source, 96–97
 licensing, 94
 limitations, 90–91
 performance and, 91–94
 philosophy, 127–128
 platform limitations, 94
 prototype, 95
 stand-alone functions/classes, 87
 strategies, 90–95, 127
 support, 94
 third-party application bundles, 95–96
 writing reusable code, 86
`reverse()` functions, 62, 494
reverse iterators, 629–630
`reverse_iterator` class, 629, 629–630
reversible containers, 658
ring buffers, 835–839
round-robin scheduling with vector, 486–490
RTTI (Run Time Type Information), 268–269
run-time debug mode, 835
`runtime_error` exception, 380
RUP (Rational Unified Process) of software development, 787–789
rvalue references, 281–286
 move assignment operator, 419

S

sample programs, employee records system, 37–46
`scanf()`, 345
scope, resolution, 302–303
scope resolution operator, 147
Scrum, 790–792
search algorithms, 458, 551–553
`seek()` method, 363–365
self-assignment, 186
self-documenting comments, 66
semantics. *See* move semantics
semaphores, initializing, 371
sequencing operator, 415
sequential containers, 468
 array class, 497–498
 deque, 491
 `forward_list`, 495–497
 `hashmap`, 670–671
 list, 492–495
 adding/removing elements, 492
 element access, 492
 enrollment, 494–495
 `forward_list` comparison, 95–497
 iterators, 492
 `merge()` method, 494
 `remove()` method, 494
 `remove_if()` method, 494
 `reverse()` method, 494
 size, 492
 `sort()` method, 494
 special operations, 493–495
 `unique()` method, 494
 vector, type parameters, 473
`set()` method, 196
set algorithms, 566–569
set associative container, 515–516
`setjmp()`, 371
`setValue()` method, 145
SFINAE (Substitution Failure Is Not An Error), 703–705
shallow copying, 180
`shared_ptr`, 446, 731–732
short variable, 9

short-circuit logic, 18
`sizeof()` function, 49
slicing, 235
smart pointers, 28–29, 727–734
 `auto_ptr`, 728
 C++ standard library, 446
 classes, writing, 723–734
 double deletion and, 732–733
 reference counting, 732–733
 `shared_ptr`, 731–732
 stack unwinding and, 396
 `unique_ptr`, 729–731
 `weak_ptr`, 734
smart pointers, writing, 723–734
software, life cycle
 RUP (Rational Unified Process), 787–789
 Spiral Model, 785–787
 Stagewise Model, 783
 Waterfall Model, 783–785
software design, 80
 feature creep, 782
software engineering, 781
 Agile Methodology, 789
 Scrum, 790–792
 XP (Extreme Programming), 792–796
software triage, 796
`sort()` function, 565–566
`sort()` method, 494
sorting algorithms, 462, 565–566
Source Code Control software, 797–799
source files, template definitions, 325
source ranges, 556
spaces in code, 74
specialization
 function templates, 340–341
 partial specialization
 classes, 679–683
 functions, 683–684
splicing, 492
Spreadsheet class
 definition, 178
Spreadsheet class, 144
 assignment operator, 185–187
 copy constructors, declaration, 185

SpreadsheetCell, 144
SpreadsheetCell base class, 237–239
SpreadsheetCell class, 144

- subclassing, 240

spreadsheets, 144

- application example
 - Spreadsheet** class, 144
 - SpreadsheetCell** base class, 237–239
 - SpreadsheetCell** class, 144
- inheritance and, polymorphism, 236–237

stack, 25

- constructors on, 154
- objects on, 151
 - destruction, 169
 - out of scope, 169
- unwinding, 394–397
- variables, exceptions, 31

stack arrays, multi-dimensional, 715–716
stack container, 451
stack container adapter

- example, 504
- operations, 504

stack frame, 25
stand-alone, reuse and, 87
stand-alone functions/classes, reuse and, 87
standard library, 36

- classes, 36

standard output, 348
start-time debug mode, 831–834
statements

- catch**, 388–389, 404
- class definitions, 144
- throw**, 373

static data members, 188–190

- access, 189–190
- counters, 189

static keyword

- data members, 291
- functions, 293–294
- linkage, 291–292
- methods, 291
- variables, 293–294

static linkage, 291
static methods, 193–194

overridden methods, 257–258
static_assert macro, 840–841
static_cast, 298–299
std::array, 20
std::bind(), 546–548
std::call_once(), 763–764
std::cerr, 6
std::cout, 6
std::endl, 7
std::function, 601–603

- function pointers, implementing, 602

std::pair class, 621–624
std::swap(), 647
std::vector, 36–37

- std::vector** class, 36–37

STL (standard template library), 94–95, 132, 443

- algorithms, 456–458
- comparison, 459
- heap, 463
- modifying, 459–460
- non-modifying, 458–459
- numerical processing, 464
- operational, 461
- search algorithms, 458
- sorting, 462
- utility algorithms, 459

Allocator type, 628
containers, 448–449

- array**, 450
- bitset**, 453–454
- deque**, 450
- forward_list**, 450
- hash tables, 453
- list**, 450
- map**, 452–453
- multimap**, 452
- multiset**, 452
- priority_queue**, 451
- set**, 452
- stack**, 451–452
- table listing, 454–456
- unordered associative containers, 453
- vector**, 449
- writing, 636–644

extending, algorithm writing, 634–638
iterator adapters
 insert iterators, 631–632
 move iterators, 632–633
 reverse iterators, 629–630
 stream iterators, 630–631
iterator traits, 635–636
omissions, 465
 vector container, 280–281
`strcat()` function, 49
`strcpy()` function, 48
stream iterators, 630–631
 input stream iterator, 630
 output stream iterators, 630
streams, 345, 346–347, 524. *See also* I/O
 streams
 buffered, 346
 `cout`, 348
 current position, 347
 destinations, 347
 file streams, 362–363
 `ifstream` destructor, 362
 `ofstream` destructor, 362
 `seek()` method, 363–365
 `tell()` method, 363–365
 `fstream` class, 367
in-memory, 360–361
input, 353–354
 error handling, 357–358
 `get()` method, 354–355
 `getline()`, 357
 `peek()`, 356–357
 `putback()`, 356
 `readName()` function, 354
 `unget()`, 355–356
linking, 365–366
output
 buffered, 349
 `flush()` method, 349
 `put()` method, 349
 `write()` method, 349
sources, 347
string streams, 360–362
unbuffered, 346
writing to, mutex classes, 764–765
 string class, 51–53, 445
 numeric conversion, 53–54
 operators, 51–53
string literals, 50
 `const` characters, 50
 localizing, 576
 raw string literals, 587–588
string streams, 360–362
string type, 30
strings, 445, 524
 C-style, 30
 dynamic
 C-style strings, 48–50
 raw string literals, 54–55
 string class, 51–53
 string literals, 50
 nonstandard, 30
 numeric conversions, 53–54
 underallocating, 734–735
 wide characters, 576–577
`StringSpreadsheetCell`, 239
`StringSpreadsheetCell` class, 239
`stringstream` class, 367
`stringToDouble()` method, 193
`strlen()` function, 49
strongly typed enumerations, 14–15
Stroustrup, Bjarne, 4
structs, 15, 146
structures, pointers to, 27
subclasses
 copy constructors, 264–265
 `equals` operator, 264–265
 inheritance, reuse and, 226–228
 parent classes, parent constructors, 229–230
 protected methods, 220
 public methods, 220
 replacing functionality, 228
 `SpreadsheetCell` class, 240
subclassing, 114
`SubObject` class, 403–404
subscripting operator, 423–428
 non-integral array indices, 427–428
 read-only access, 426–427

substitution pattern, 582
subsystems, code reuse and, 130–131
subtraction operator (-), 12
`subtract_with_carry_engine`, 615
superclasses, 115
 abstract, 115
 inheritance, 218
 methods
 access level, 261–263
 arguments, 260–261
 private, 259–260
`swap()` function, 482, 647
switch statements, 16–17
symbolic debugger, 843
`system_clock`, 610

T

tabs in code, 74
`tell()` method, 363–365
template classes, writing, 317–324
template metaprogramming
 factorials at compile time, 696
 loop unrolling, 696–697
 tuple printing, 698–699
 type traits, 699–702
template specializations, 334–336
 inheritance comparison, 337
template template parameters, 676–678
template types, 343
templates, 316, 444
 aliases, 338
 benefits, 133
 class templates, 317
 friend function templates, 342–343
 partial specialization, 679–683
 classes, subclassing, 336–337
 code reuse, 132–134
 coding without, 317–320
 cons, 133
 function templates, 339–343
 overloading, 341–342
 specialization, 340–341

generic programming, 444
generic template class, 133
grid class, 320–322
`hashmap` class, 637–638
header files, 325–326
inheritance comparison, 133–134
instantiation, 321
 selective, 325
iterator template arguments, 551
keywords
 `auto`, 689–691
 `decltype`, 689–691
 `make_pair()`, 505
method templates, 330–334
names, 321
`NGrid` template class, 686–689
parameterization, 316
parameters
 integer non-type parameters, 327–329
 non-type, 327–329, 678–679
 parameter packs, 694
 type, 674–676
processing, 325
recursion, 685–689
source files, 326
templated types (regex library), 588
types, 325
 zero-initialization, 328–333
variadic, 691–695
 mixin classes, 694–695
 type-safe variable-length argument lists, 692–694
 writing, 317–324
ternary operators, 11, 17
test workflow, 788
`testCallback()` function, 541
The C++ Programming Language (Bjarne), 4
The C Programming Language (Kernighan and Ritchie), 4
third-party applications, bundling, reuse and, 95–96
`this` pointer, 150–151
thread pools, 776–777

`threadFunc()` function, 753

threading

 design, 777–778

 library, 901

 models, 101

threads

 cancelling, 752

 function objects, 748–750

 function pointers, 746–748

 lambda expression, 750

 member functions, 751

 results retrieval, 752

`throw` keyword, 374

throw lists, 382–387

 usefulness, 386–387

throw statement, 373

throwing exceptions

 multiple, 378–380

 stack unwinding and, 396–397

`time_point` class, 610, 611–612

`time_point` type, 610

tokenizing values, 353

try/catch construct, 373

tuples, 447, 621–624

 printing, 698–699

type inference

`auto` keyword, 24

`decltype` keyword, 689–691

type parameters, templates, 674–676

type traits, 699–702

typedefs, 295–296

 associative containers, 658–659

 container requirements, 648

 function pointers, 296–297

`hashmap` class, 648–649

 iterators, 472–473, 655–656

`regex` library, 588–589

`typeid` operator, virtual methods, 268

`typename` keyword, 641

types, 316

 aliases, 297

 atomic types, 755–757

 categories, 699–702

 covariant return types, 249

 enumerated, 13–14

classes, 201

 strongly typed enumerations, 14–15

operator overloading and, 211

overridden methods, 249–250

relations, 702–703

string, 30

structs, 15

templates, 325

type-safe enumerations, 14–15

type-safe variable-length argument lists, 692–694

U

UCS (Universal Character Set), 577

unary operators, 11

 overloading, 420

unbuffered streams, 346

unexpected() function, 384

unexpected_handler, 384

unget() method, 355–356

uniform initialization, 476

uniform random number generator, 447

uninitialized pointers, dereferencing, 27

unions, 146

unique() function, 494, 562

unique_ptr, 446, 729–731

unit testing

 sample data, 912–913

unordered associative containers, 453,

 516–522, 636–644. *See also* hash tables

 unordered_map, 519–521

 example, 521–522

 map comparison, 519–520

 unordered_multimap, 522

 unordered_multiset, 522

 unordered_set, 522

unordered_map associative container,

 519–521

 example, 521–522

 map comparison, 519–520

unordered_multimap associative

 container, 522

unordered_multiset associative

 container, 522

unordered_set associative container, 522
unsigned int variable, 9
unsigned long long variable, 9
unsigned long variable, 9
unsigned short variable, 9
unwinding stack, 394–397
upcasting, 235
 ambiguities and, 247
upper_bound() function, 566
user interface, dependencies, code reuse
 and, 132
user-defined, vectors, 476
user-defined classes, vectors, 476
user-defined literals (C++11), 307–309
using directive, 8
using keyword, 253
utilities
 mathematical, 446–447
 time, 447
utility algorithms, 459, 555–556
utilities, variable-length argument lists,
 310–312

V

valarray class, 446
value semantics, 468–469
values, 316
 parameterizing, 316
 passing by, 277
 disallowing, 188
 passing variables by, 27
 return values, references, 278
 returning, 22–23
variable-length argument lists, 310–312
variables, 9
 bool, 10
 casting, 10
 char, 10
 char16_t, 10
 char32_t, 10
 coercing, 10
 condition variables, 767–769
 const keyword, 287
 converting, 10

declaring, 9
double, 10
float, 10
int, 10
long, 10
long double, 10
long long, 10
nonlocal, order of initialization, 294
passing
 by reference, 27, 277
 by value, 27
 by values, 277
pointers, 27
reference variables, 274–276
short, 10
stack, exceptions, 31
static, 293–294
string literals, 50
type changing, 10
unsigned int, 10
unsigned long, 10
unsigned long long, 10
unsigned short, 10
wchar_t, 10
variadic templates, 691–695
mixin classes, 694–695
ring buffer and, 835–839
type-safe variable-length argument lists,
 692–694
vector, 283
 assigning, 476–477
 built-in classes, 476
 comparing, 477
 constructors, 475–476
 copying, 476–477
 destructors, 75–476
 dynamic-length, 475
 elements
 access methods, 474–475
 appending, 481–485
 initial value, 474–475
 user-defined classes, 476
 fixed-length, 474–475
 heap, allocation, 476
 intVector, 476–477

iterators, 478–481
memory allocation scheme, 485–486
`printVector()` function, 482
`push_back()` method, 372
round-robin scheduling, 486–490
`vector<bool>`, 490–491

vector
type parameters, 473

vector container, 280–281, 449–450

virtual base classes, 248, 270–271

virtual destructors, 232
method templates, 330
need for, 267–268

virtual keyword, 222
implementing, 266–267
justification for, 267

virtual methods, 222
method templates, 330
overriding, 265–266
pure, 238–239
`typeid` operator, 268

Visual C++ 2010 debugger, 859–861
voter registration auditing algorithm example, 570–573

W

`wait()` method, 768
`wchar_t` variable, 10
`weak_ptr`, 734
WeatherPrediction class, 225–228
 MyWeatherPrediction class, 225–228
`what()` method
 overriding, 340
Windows, memory leaks, 737–738
word boundaries, 586
workflows, 788
`write()` method, 349

X-Y-Z

zero-initialization, 476

Try Safari Books Online FREE for 15 days and take 15% off for up to 6 Months*

Gain unlimited subscription access to thousands of books and videos.



With Safari Books Online, learn without limits from thousands of technology, digital media and professional development books and videos from hundreds of leading publishers. With a monthly or annual unlimited access subscription, you get:

- Anytime, anywhere mobile access with Safari To Go apps for iPad, iPhone and Android
- Hundreds of expert-led instructional videos on today's hottest topics
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Rough Cuts pre-published manuscripts

START YOUR FREE TRIAL TODAY!
Visit: www.safaribooksonline.com/wrox

*Discount applies to new Safari Library subscribers only and is valid for the first 6 consecutive monthly billing cycles. Safari Library is not available in all countries.



An Imprint of  WILEY
Now you know.



Programmer to Programmer™

Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ p2p.wrox.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Wrox.com

Browse the vast selection of Wrox titles, e-books, and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

Wrox on

Join the Wrox Facebook page at facebook.com/wroxpress and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support? Let us know by e-mailing wrox-partnerwithus@wrox.com

Related Wrox Books

Beginning ASP.NET 4.5.1

ISBN: 978-1-118-84677-3

Beginning ASP.NET 4.5.1 guides you through the process of creating a fully functional, database-driven website, from creation of the most basic site structure all the way down to the successful deployment of the website to a production environment. Best-selling author Imar Spaanjaars provides a firm foundation for coders new to ASP.NET and key insights for those not yet familiar with the important updates in the 4.5.1 release.

Ivor Horton's Beginning Visual C++ 2013

ISBN: 978-1-118-84571-4

This guide delivers a comprehensive introduction to both the standard C++ language and Visual C++ 2013. Step-by-step programming exercises, helpful examples, and clear solutions deftly walk you through the ins and outs of the latest C++ development.

Professional C# 5.0 and .NET 4.5.1

ISBN: 978-1-118-83303-2

Written by a dream team of .NET experts, this book includes everything developers need to work with C#, the language of choice for .NET applications. *Professional C# 5.0 and .NET 4.5.1* is perfect for both experienced C# programmers looking to sharpen their skills and professional developers who are using C# for the first time.

Professional Visual Studio 2013

ISBN: 978-1-118-83204-2

This comprehensive guide provides a complete view of the new and updated tools featured in Visual Studio 2013 that make programming easier and more efficient. Topics include new shortcuts, UI testing capabilities, and customization options, along with changes to target SharePoint 2013, advanced debugging capabilities, the Data Compare function, and a host of others.

Professional Application Lifecycle Management with Visual Studio 2013

ISBN: 978-1-118-83658-3

Microsoft's Application Lifecycle Management (ALM) makes software development easier and now features support for iOS, MacOS, Android, and Java development. Microsoft's ALM suite of productivity tools includes new functionality and extensibility that are sure to grab your attention. *Professional Application Lifecycle Management with Visual Studio 2013* provides in-depth coverage of these new capabilities.

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.