

An Introduction to Design Patterns in C++ with Qt 4

BRUCE PERENS' OPEN SOURCE SERIES

www.prenhallprofessional.com/perens

Bruce Perens' Open Source Series is a definitive series of books on Linux and open source technologies, written by many of the world's leading open source professionals. It is also a voice for up-and-coming open source authors. Each book in the series is published under the Open Publication License (www.opencontent.org), an open source compatible book license, which means that electronic versions will be made available at no cost after the books have been in print for six months.

- ◆ *Java™ Application Development on Linux®*
Carl Albing and Michael Schwarz
- ◆ *C++ GUI Programming with Qt 3*
Jasmin Blanchette and Mark Summerfield
- ◆ *Managing Linux Systems with Webmin: System Administration and Module Development*
Jamie Cameron
- ◆ *User Mode Linux®*
Jeff Dike
- ◆ *An Introduction to Design Patterns in C++ with Qt 4*
Alan Ezust and Paul Ezust
- ◆ *Understanding the Linux Virtual Memory Manager*
Mel Gorman
- ◆ *PHP 5 Power Programming*
Andi Gutmans, Stig Bakken, and Derick Rethans
- ◆ *Linux® Quick Fix Notebook*
Peter Harrison
- ◆ *Implementing CIFS: The Common Internet File System*
Christopher Hertel
- ◆ *Open Source Security Tools: A Practical Guide to Security Applications*
Tony Howlett
- ◆ *Apache Jakarta Commons: Reusable Java™ Components*
Will Iverson
- ◆ *Linux® Patch Management: Keeping Linux® Systems Up To Date*
Michael Jang
- ◆ *Embedded Software Development with eCos*
Anthony Massa
- ◆ *Rapid Application Development with Mozilla*
Nigel McFarlane
- ◆ *Subversion Version Control: Using the Subversion Version Control System in Development Projects*
William Nagel
- ◆ *Intrusion Detection with SNORT: Advanced IDS Techniques Using SNORT, Apache, MySQL, PHP, and ACID*
Rafeeq Ur Rehman
- ◆ *Cross-Platform GUI Programming with wxWidgets*
Julian Smart and Kevin Hock with Stefan Csomor
- ◆ *Samba-3 by Example, Second Edition: Practical Exercises to Successful Deployment*
John H. Terpstra
- ◆ *The Official Samba-3 HOWTO and Reference Guide, Second Edition*
John H. Terpstra and Jelmer R. Vernooij, Editors
- ◆ *Self-Service Linux®: Mastering the Art of Problem Determination*
Mark Wilding and Dan Behman
- ◆ *AJAX: Creating Web Pages with Asynchronous JavaScript and XML*
Edmond Woychowsky

An Introduction to Design Patterns in C++ with Qt 4

Alan Ezust
Paul Ezust



PRENTICE
HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com



This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to <http://www.prenhallprofessional.com/safarienabled>
- Complete the brief registration form
- Enter the coupon code 1AMQ-56YL-KJ1A-LXLU-GGLT

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: www.prenhallprofessional.com

Library of Congress Cataloging-in-Publication Data

Ezust, Alan.

An introduction to design patterns in C++ with Qt 4 / Alan Ezust, Paul Ezust.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-187905-7 (pbk. : alk. paper)

1. C++ (Computer program language) 2. Software patterns. 3. Computer software—Reusability.
I. Ezust, Paul. II. Title.

QA76.73.C153E94 2006

005.13'3—dc22

2006011947

Copyright © 2007 Pearson Education, Inc.

This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, v1.0 or later. (The latest version is presently available at <http://www.opencontent.org/openpub/>.)

For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458
Fax: (201) 236-3290

ISBN 0-13-187905-7

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, August 2006

*This book is dedicated to Miriam Ezust,
without whom none of our work
would have been possible.*

Contents

<i>Preface</i>	<i>xix</i>
<i>Acknowledgments</i>	<i>xxiii</i>
<i>Rationale for the Book</i>	<i>xxv</i>
<i>About the Authors</i>	<i>xxvii</i>

PART I Introduction to C++ and Qt 4 2

1	C++ Introduction	5
1.1	Overview of C++	6
1.2	A Brief History of C++	6
1.3	Setup: Open-Source Platforms	7
1.3.1	<i>*nix</i>	7
1.3.2	<i>Downloading from Source</i>	9
1.4	Setup: Win32	12
1.5	C++ First Example	12
1.6	Input and Output	16
1.7	Identifiers, Types, and Literals	19
1.8	C++ Simple Types	22
1.8.1	<i>main and Command Line Arguments</i>	24
1.8.2	<i>Arithmetic</i>	25
1.9	C++ Standard Library Strings	30
1.10	Streams	31
1.11	The Keyword <code>const</code>	34
1.12	Pointers and Memory Access	36
1.12.1	<i>The Unary Operators <code>&</code> and <code>*</code></i>	36
1.12.2	<i>Operators <code>new</code> and <code>delete</code></i>	38

- 1.13 `const*` and `*const` 40
- 1.14 Reference Variables 43
 - Points of Departure 44
 - Review Questions 45

- 2 Classes 47**
 - 2.1 Structs 48
 - 2.2 Class Definitions 49
 - 2.3 Member Access Specifiers 51
 - 2.4 Encapsulation 54
 - 2.5 Introduction to UML 54
 - 2.5.1 *UML Relationships* 55
 - 2.6 Friends of a Class 55
 - 2.7 Constructors 56
 - 2.8 Subobjects 58
 - 2.9 Destructors 60
 - 2.10 The Keyword `static` 61
 - 2.11 Copy Constructors and Assignment Operators 64
 - 2.12 Conversions 67
 - 2.13 `const` Member Functions 68
 - Review Questions 79

- 3 Introduction to Qt 81**
 - 3.1 Example Project: Using `QApplication` and `QLabel` 82
 - 3.2 Makefile, `qmake`, and Project Files 83
 - 3.2.1 *#include: Finding Header Files* 85
 - 3.2.2 *The make Command* 86
 - 3.2.3 *Cleaning Up Files* 88
 - 3.3 Getting Help Online 89
 - 3.4 Style Guidelines and Naming Conventions 90
 - 3.5 The Qt Core Module 91
 - 3.6 Streams and Dates 91
 - Points of Departure 93
 - Review Questions 94

-
- 4 Lists 95**
 - 4.1 Introduction to Containers 96
 - 4.2 Iterators 97
 - 4.2.1 *QStringList and Iteration* 97
 - 4.3 Relationships 99
 - Points of Departure 102
 - Review Questions 103

 - 5 Functions 105**
 - 5.1 Function Declarations 106
 - 5.2 Overloading Functions 107
 - 5.3 Optional Arguments 109
 - 5.4 Operator Overloading 111
 - 5.5 Parameter Passing by Value 116
 - 5.6 Parameter Passing by Reference 118
 - 5.7 References to `const` 121
 - 5.8 Function Return Values 122
 - 5.9 Returning References from Functions 122
 - 5.10 Overloading on `const`-ness 124
 - 5.11 Inline Functions 126
 - 5.12 Inlining versus Macro Expansion 127
 - Review Questions 133

 - 6 Inheritance and Polymorphism 135**
 - 6.1 Simple Derivation 136
 - 6.1.1 *Inheritance Client Code Example* 141
 - 6.2 Derivation with Polymorphism 142
 - 6.3 Derivation from an Abstract Base Class 148
 - 6.4 Inheritance Design 152
 - 6.5 Overloading, Hiding, and Overriding 154
 - 6.6 Constructors, Destructors, and Copy Assignment Operators 155
 - 6.7 Processing Command-Line Arguments 158
 - 6.7.1 *Derivation and ArgumentList* 159
 - Points of Departure 164
 - Review Questions 165

PART II Higher-Level Programming 166

7 Libraries 169

- 7.1 Code Containers 170
- 7.2 Reusing Other Libraries 171
- 7.3 Organizing Libraries: Dependency Management 173
 - 7.3.1 *Installing Libraries* 176
- 7.4 Installing Libraries: A Lab Exercise 176
 - 7.4.1 *Fixing the Linker Path* 177
- 7.5 Frameworks and Components 178
 - Review Questions 180

8 Introduction to Design Patterns 181

- 8.1 Iteration and the Visitor Pattern 182
 - 8.1.1 *Directories and Files: QDir and QFileInfo* 183
 - 8.1.2 *Visitor Pattern* 184
 - 8.1.3 *Customizing the Visitor Using Inheritance* 186
 - Review Questions 190

9 QObject 191

- 9.1 QObject's Child Management 194
- 9.2 Composite Pattern: Parents and Children 196
 - 9.2.1 *Finding Children* 199
- 9.3 QApplication and the Event Loop 200
 - 9.3.1 *Layouts: A First Look* 202
 - 9.3.2 *Connecting to Slots* 203
 - 9.3.3 *Signals and Slots* 204
- 9.4 Q_OBJECT and moc: A Checklist 209
- 9.5 Values and Objects 210
- 9.6 tr() and Internationalization 211
 - Point of Departure 211
 - Review Questions 212

-
- 10 Generics and Containers 213**
 - 10.1 Generics and Templates 214
 - 10.1.1 Function Templates 214*
 - 10.1.2 Class Templates 216*
 - 10.2 Containers 219
 - 10.3 Managed Containers, Composites, and Aggregates 221
 - 10.4 Implicitly Shared Classes 224
 - 10.5 Generics, Algorithms, and Operators 225
 - 10.6 Serializer Pattern 227
 - 10.6.1 Benefits of the Serializer 229*
 - 10.7 Sorted Map Example 229
 - Review Questions 235

 - 11 Qt GUI Widgets 237**
 - 11.1 Widget Categories 239
 - 11.2 QMainWindow and QSettings 240
 - 11.2.1 QSettings: Saving and Restoring Application State 242*
 - 11.3 Dialogs 244
 - 11.3.1 Input Dialogs and Widgets 246*
 - 11.4 Images and Resources 248
 - 11.5 Layout of Widgets 251
 - 11.5.1 Spacing, Stretching, and Struts 254*
 - 11.5.2 Moving Widgets across Layouts 256*
 - 11.6 QActions, QMenus, and QMenuBar 260
 - 11.7 QActions, QToolbars, and QActionGroups 262
 - 11.7.1 The Command Pattern 262*
 - 11.8 Regions and QDockWidgets 270
 - 11.9 Views of a QStringList 272
 - Points of Departure 274
 - Review Questions 275

-
- 12 Concurrency 277**
 - 12.1 QProcess and Process Control 278
 - 12.1.1 *Processes and Environment* 280
 - 12.1.2 *Qonsole: Writing an Xterm in Qt* 284
 - 12.1.3 *Qonsole with Keyboard Events* 286
 - 12.2 Threads and QThread 290
 - 12.2.1 *QPixmap and QThread Animation*
 - Example: Movie Player* 290
 - 12.2.2 *Movie Player with QTimer* 294
 - 12.2.3 *Multiple Threads, Queues, and Loggers*
 - Example: Giant* 296
 - 12.2.4 *Thread Safety and QObjects* 302
 - 12.3 Summary: QProcess and QThread 303
 - Review Questions 305

 - 13 Validation and Regular Expressions 307**
 - 13.1 Validators 308
 - 13.2 Regular Expressions 310
 - 13.2.1 *Regular Expression Syntax* 311
 - 13.2.2 *Regular Expressions: Phone Number Recognition* 313
 - 13.3 Regular Expression Validation 316
 - Review Questions 319

 - 14 Parsing XML 321**
 - 14.1 The Qt XML Module 325
 - 14.2 Event-Driven Parsing 325
 - 14.3 XML, Tree Structures, and DOM 329
 - 14.3.1 *Visitor Pattern: DOM Tree Walking* 331
 - 14.3.2 *Generation of XML with DOM* 335
 - Review Questions 340

 - 15 Meta Objects, Properties, and Reflective Programming 341**
 - 15.1 Anti-patterns 342
 - 15.2 QMetaObject: The MetaObject Pattern 344

- 15.3 Type Identification and `QObject::cast` 345
- 15.4 `Q_PROPERTY` Macro: Describing `QObject` Properties 347
- 15.5 `QVariant` Class: Accessing Properties 350
- 15.6 `DataObject`: An Extension of `QObject` 353
- 15.7 Property Containers: `PropsMap` 355
 - Review Questions 357

- 16 More Design Patterns 359**
 - 16.1 Creational Patterns 360
 - 16.1.1 *Abstract Factory* 361
 - 16.1.2 *Abstract Factories and Libraries* 363
 - 16.1.3 *qApp and Singleton Pattern* 365
 - 16.1.4 *Creation Rules and friend Functions*
(What Friends Are Really For) 366
 - 16.1.5 *Benefits of Using Factories* 369
 - 16.2 Serializer Pattern Revisited 373
 - 16.2.1 *Exporting to XML* 375
 - 16.2.2 *Importing Objects with an Abstract Factory* 376
 - 16.3 The Façade Pattern 381
 - 16.3.1 *Functional Façade* 384
 - 16.3.2 *Smart Pointers: auto_ptr* 384
 - 16.3.3 *FileTagger: Façade Example* 385
 - Points of Departure 389
 - Review Questions 390

- 17 Models and Views 391**
 - 17.1 M-V-C: What about the Controller? 392
 - 17.2 Dynamic Form Models 393
 - 17.2.1 *Form Models* 397
 - 17.2.2 *Form Views* 400
 - 17.2.3 *Unforeseen Types* 403
 - 17.2.4 *Controlling Actions* 404
 - 17.2.5 *DataObject Form Model* 405
 - 17.3 Qt 4 Models and Views 409
 - 17.4 Table Models 411

- 17.5 Tree Models 417
 - 17.5.1 *Extended Tree Widget Items* 418
 - Review Questions 421

- 18 Qt SQL Classes 423**
 - 18.1 Introduction to MySQL 424
 - 18.2 Queries and Result Sets 427
 - 18.3 Database Models 429
 - Review Questions 433

- PART III C++ Language Reference 434**

- 19 Types and Expressions 437**
 - 19.1 Operators 438
 - 19.2 Evaluation of Logical Expressions 443
 - 19.3 Enumerations 443
 - 19.4 Signed and Unsigned Integral Types 445
 - 19.5 Standard Expression Conversions 447
 - 19.6 Explicit Conversions 449
 - 19.7 Safer Typecasting Using ANSI C++ Typecasts 450
 - 19.7.1 *static_cast and const_cast* 450
 - 19.7.2 *reinterpret_cast* 453
 - 19.7.3 *Why Not Use C-style Casts?* 454
 - 19.8 Run-Time Type Identification (RTTI) 454
 - 19.8.1 *typeid Operator* 456
 - 19.9 Member Selection Operators 457
 - Point of Departure 458
 - Review Questions 461

- 20 Scope and Storage Class 463**
 - 20.1 Declarations and Definitions 464
 - 20.2 Identifier Scope 465
 - 20.2.1 *Default Scope of Identifiers: A Summary* 466
 - 20.2.2 *File Scope versus Block Scope and operator::* 468
 - 20.3 Storage Class 470
 - 20.3.1 *Globals, statics, and QObject* 471

- 20.4 Namespaces 473
 - 20.4.1 *Anonymous Namespaces* 476
 - 20.4.2 *Open Namespaces* 476
 - 20.4.3 *Namespace, static Objects and extern* 476
 - Review Questions 478

21 Statements and Control Structures 479

- 21.1 Statements 480
- 21.2 Selection Statements 480
- 21.3 Iteration 483
- 21.4 Exceptions 485
 - 21.4.1 *Exception Handling* 486
 - 21.4.2 *Exception Types* 486
 - 21.4.3 *throwing Things Around* 487
 - 21.4.4 *try and catch* 490
 - 21.4.5 *More about throw* 494
 - 21.4.6 *Rethrown Exceptions* 496
 - 21.4.7 *Exception Expressions* 497
 - Review Questions 502

22 Memory Access 503

- 22.1 Pointer Pathology 504
- 22.2 Further Pointer Pathology with Heap Memory 506
- 22.3 Memory Access Summary 509
- 22.4 Introduction to Arrays 509
- 22.5 Pointer Arithmetic 510
- 22.6 Arrays, Functions, and Return Values 511
- 22.7 Different Kinds of Arrays 513
- 22.8 Valid Pointer Operations 513
- 22.9 What Happens If new Fails? 515
 - 22.9.1 *set_new_handler(): Another Approach to New Failures* 516
 - 22.9.2 *Using set_new_handler and bad_alloc* 517
 - 22.9.3 *Checking for null: The Updated Way to Test for New Failures* 518
- 22.10 Chapter Summary 519
 - Review Questions 521

23 Inheritance in Detail 523

- 23.1 Virtual Pointers and Virtual Tables 524
- 23.2 Polymorphism and virtual Destructors 526
- 23.3 Multiple Inheritance 528
 - 23.3.1 *Multiple Inheritance Syntax* 529
 - 23.3.2 *Multiple Inheritance with Abstract Interfaces* 531
 - Point of Departure 532
 - 23.3.3 *Resolving Multiple Inheritance Conflicts* 532
- 23.4 public, protected, and private Derivation 536
 - Review Questions 539

24 Miscellaneous Topics 541

- 24.1 Functions with Variable-Length Argument Lists 542
- 24.2 Resource Sharing 543

PART IV Programming Assignments 548**25 MP3 Jukebox Assignments 551**

- 25.1 Data Model: Mp3File 553
- 25.2 Visitor: Generating Playlists 555
- 25.3 Preference: An Enumerated Type 556
- 25.4 Reusing id3lib 559
- 25.5 PlaylistModel Serialization 560
- 25.6 Testing Mp3File Related Classes 561
- 25.7 Simple Queries and Filters 561
- 25.8 Mp3PlayerView 563
- 25.9 Models and Views: Playlist 565
- 25.10 Source Selector 566
- 25.11 Persistent Settings 567
- 25.12 Edit Form View for FileTagger 568
- 25.13 Database View 569
 - Points of Departure 571

PART V Appendices 572

Appendix A: C++ Reserved Keywords 575

Appendix B: Standard Headers 577

Appendix C: The Development Environment 579

C.1 The Preprocessor: For #including Files 579

C.2 Understanding the Linker 582

C.2.1 Common Linker Error Messages 584

C.3 Debugging 587

C.3.1 Building a Debuggable Target 588

C.3.2 gdb Quickstart 589

C.3.3 Finding Memory Errors 590

C.4 Qt Assistant and Designer 593

C.5 Open-Source IDEs and Development Tools 594

C.5.1 UML Modeling Tools 597

C.5.2 jEdit 598

Bibliography 601

Index 603

Preface

C++ had been in use for many years before it was standardized in 1989, which makes it a relatively mature language compared to others that are in popular use today. It is a very important language for building fast, efficient, mission-critical systems. C++ is also one of the most flexible languages around, giving developers many choices of programming styles for use in high-level GUI code as well as low-level device drivers.

For a few years in the early '90s, C++ was the most popular object-oriented (OO) language in use, and many computer science (CS) students were introduced to object-oriented programming (OOP) via C++. This was because C++ provided a relatively easy transition to OOP for C programmers, and many CS professors had been teaching C previously.

Starting around 1996, Java gained favor over C++ as the first OO language for students to learn. There are a number of reasons that Java gained so much popularity.

- The language itself is simpler than C++.
- The language has built-in garbage collection, so programmers do not need to concern themselves with memory de-allocation.
- A standard set of GUI classes is included in the development kit.
- The built-in `String` class supports Unicode.
- Multithreading is built into the language.
- It is easier to build and “plug in” Java Archives (JARs) than it is to recompile and relink libraries.
- Many Web servers provide Java APIs for easy integration.
- Java programs are platform independent (Wintel, Solaris, MacOS, Linux, *nix, etc.).

Many of Java's benefits listed above can be achieved with C++ used in conjunction with Qt 4.

- Qt provides a comprehensive set of GUI classes that run faster, look better, and are more flexible than Java's `Swing` classes.
- Signals and slots are easier to use than `(Action|Event|Key)Listener` interfaces in Java.
- Qt 4 has a plugin architecture that makes it possible to load code into an application without recompiling or relinking.
- Qt 4 provides `foreach`, which makes iteration through collections simpler to read and write.

Although Qt does not provide garbage collection, there are a variety of alternatives one can use to avoid the need to delete heap objects directly.

1. Containers (see Section 10.2)
2. Parents and children (see Section 9.2)
3. `auto_ptr` (see Section 16.3.2)
4. `QPointer` (see Section 19.9).
5. Subobjects (see Section 2.8)
6. Stack objects (see Section 20.3)

Using C++ with Qt comes very close to Java in ease of use, comprehensiveness, and convenience. It significantly exceeds Java in the areas of speed and efficiency, making everything from processing-intensive server applications to high-speed graphics-intensive games possible.

Another benefit of learning C++ with Qt comes from Qt's widespread use in open-source projects. There is already a great wealth of free open-source code that you can learn from, reuse, and perhaps help to improve.

How to Use This Book

Part I contains an introduction to C++, UML, and the Qt core. This part is designed to avoid forward referencing as much as possible, and it presents the topics in an order and a level of detail that should not overwhelm someone who is new to C/C++.

In Part II, you will find higher-level programming ideas, Qt modules, and design patterns. Here we present paradigm-shifting ways of writing code and organizing objects in a modular fashion.

For completeness and for reference, Part III covers in more depth some of the “dry” but important C++ features that were introduced in Part I. By the time the reader has reached this point, these ideas should be a lot easier to understand.

At the end of each chapter, you will find exercises and review questions. Most of the programming exercises have solutions available on our Web site. For the questions, if the answers are not in the preceding chapter, then often there are pointers on where to find them. If this book is used for a course, these questions could be asked by the student or by the teacher, in the classroom or on an exam.

Source code files for all the examples in this book are contained in the file `src.tar.gz`, which can be downloaded from <http://oop.mcs.suffolk.edu/dist>.

A Note about Formats and Book Production

What you are reading now is only one of a number of possible versions of this text available. Because the document was originally written in XML, using a “literal programming” style, we can generate a variety of different versions (bulleted slides, paragraphed textbook, with or without solutions, etc.) in a variety of different formats (`html`, `pdf`, `ps`, `htmlhelp`).

Each programming example is extracted from working source code. The Web version provides a hyperlink from each code excerpt to its full source file. This makes it very easy to try the examples yourself. The text and listings in the Web version also contain hyperlinks from each library `ClassName` to its class documentation page.

We wrote the original manuscript using `jEdit` and `gnu-emacs`, marking it up with a modified `DocBook/XML` syntax that we converted into pure `DocBook/XML` using a custom XML processor called Slacker’s `DocBook` written in Python. Most of the original diagrams were produced with `Umbrello` or `Dia`, but a couple were produced with `Doxygen` and `Dot`. The final book was typeset in `QuarkXPress`. We generate many different versions (overhead slides, textbooks, labs, and solutions) of this text for our own use, some in HTML, some in PostScript, and some in PDF.

The XML and image processors we used were `Apache Ant`, `Xerces`, `FOP`, `Gnu xsltproc`, `ReportLab pyRXP`, `ImageMagick`, `JAI`, `JINI`, and `XEP`. We did all of the editing and processing of the original manuscript on GNU/Linux systems under KDE. The example programs all compile and run under Linux.

The cover photo is of the Panama Canal. Before there was a Panama Canal, ships had to travel down and then up the entire length of South America to get from one coast of the United States to the other. The canal provided a much shorter and more direct path. The aim of this book is to provide a shorter and

more direct path for programmers who don't have a lot of extra time and who need to obtain working mastery of C++ OOP and design patterns. Qt 4 makes this possible.

Style Conventions

- *Monospace*—used for any literal symbol that appears in the code listings
- **Bold**—used the first time a term appears (key terms, defined terms, etc.)
- *Italic*—used for emphasis, and also used for wildcards (terms that need to be replaced by “real types” when they are actually used). In monospace text, these terms are set italic and monospace.

Acknowledgments

Thanks to the many authors and contributors involved in following open-source projects, for making the free tools, for answering questions, and for writing good docs. We used all of these programs to make this book:

- jEdit (<http://jedit.sourceforge.net>)
- Umbrello (<http://uml.sourceforge.net/index.php>)
- Firefox (<http://www.mozilla.org/products/firefox/>),
Mozilla (<http://www.mozilla.org/>)
- Doxygen (<http://www.stack.nl/~dimitri/doxygen/>)
- dia (<http://www.gnome.org/projects/dia/>)
- imagemagick (<http://www.imagemagick.org/script/index.php>)
- graphviz (<http://www.research.att.com/sw/tools/graphviz/>)
- KDE (<http://www.kde.org/>), kdevelop (<http://www.kdevelop.org/>),
kdbg (<http://members.nextra.at/johsixt/kdbg.html>)
- docbook (<http://www.docbook.org/>),
docbook-xsl (<http://docbook.sourceforge.net/projects/xsl/>)
- xsltproc, xmllint, gnu xml libs (<http://xmlsoft.org/XSLT/xsltproc2.html>)
- cvs (<http://www.nongnu.org/cvs/>),
subversion (<http://subversion.tigris.org/>)
- MoinMoin (<http://moinmoin.wikiwikiweb.de/>)
- Bugzilla (<http://www.bugzilla.org/>)
- Apache httpd (<http://httpd.apache.org/>), ant (<http://ant.apache.org/>),
fop (<http://xmlgraphics.apache.org/fop/>)
- gaim (<http://gaim.sourceforge.net>)
- Python (<http://www.python.org>)

- ReportLab PyRXP (<http://www.reportlab.org/pyrxp.html>)
- PyQt (<http://www.riverbankcomputing.co.uk/pyqt/index.php>),
pyKDE (<http://www.riverbankcomputing.co.uk/pykde/index.php>),
qscintilla (<http://www.riverbankcomputing.co.uk/qscintilla/index.php>),
eric3 (<http://www.die-offenbachs.de/detlev/eric3.html>)
- mysql (<http://www.mysql.com/>)
- GNU Emacs (<http://www.gnu.org/software/emacs/emacs.html>)
- Linux (<http://www.kernel.org/>), gcc (<http://gcc.gnu.org/>),
gdb (<http://www.gnu.org/software/gdb/gdb.html>)
- valgrind (<http://valgrind.org/>)

Thanks to Norman Walsh [docbook] and Bob Stayton [docbookxsl] for developing and documenting a superb system of publishing tools. Thanks to the rest of the docbook community for help and guidance.

Thanks to the volunteers at debian.org for keeping Sid up to date and still stable enough to be a productive development platform. Thanks to irc.freenode.net for bringing together a lot of good brains.

Thanks to Emily Ezust for wordsmithing skills and for getting us started with Qt in the first place. Thanks to the reviewers who provided input and valuable feedback on the text: Mark Summerfield, Johan Thelin, Stephen Dewhurst, Hal Fulton, David Boddie, Andy Shaw, and Jasmin Blanchette [Blanchette04], who also taught us the Qt 4 dance!

Thanks to Matthias Ettrich for the vision and motivation. Thanks to the rest of the team at Trolltech for writing good docs, answering questions on the mailing lists, and being so “open” with the open-source community.

Thanks to the editorial and production staff at Prentice Hall for their meticulous reading of our book and for helping us to find the many errors that were distributed throughout the text.

Finally, thanks to Suffolk University, a source of great support throughout this project. In particular, thanks to Andrew Teixeira for his tireless help in maintaining the servers that we used heavily for the past two years and for his invaluable technical advice. Thanks also to the students who took CMPSC 331/608 using the evolving preliminary versions of this book since the fall of 2003 and who provided us with a stream of valuable feedback.

Rationale for the Book

At Suffolk University, we buck the popular trend and continue teaching object-oriented programming using C++. For many years we taught a standard one-semester OOP/C++ course that followed a CS1-CS2 sequence based on the C programming language. Students developed substantial mastery of the core C++ language and an understanding of some OO concepts such as encapsulation, refactoring, and tool development. However, we found that STL is a library that often overwhelms students and causes them to spend too much time on low-level programming constructs and template issues. STL is not enough to write applications with graphical interfaces, and another framework would have to be used anyway.

During the summer of 2003 we got together and decided to develop a book that would approach OOP/C++ at a higher level. We wanted to provide a substantial introduction to GUI programming using the multi-platform Qt framework, as well as touch upon some important design patterns that are used.

We designed this first as a textbook to be used in a university class, but it is designed in an extensible way and crammed with information that will make it useful for readers with a wide range of backgrounds: from those who already program in C or another procedural language and wish to learn OO and GUI programming, to those who have no C background but are familiar with Basic, Java, Python, or another programming language and wish to learn C++. The first part of the book is aimed at familiarizing all audiences with basic C++ elements, OO concepts, UML, and the core Qt classes.

We believe that readers will understand ideas best when they apply them, and we found this to be especially true with design patterns. Many of the Qt classes or code examples are concrete implementations of some of the more popular design patterns described in *Design Patterns* [Gamma95]. For each design pattern that we discuss, we make available the source code for our example and include exercises that challenge readers to reuse, refine, and extend that code.

Reuse of libraries requires an understanding not only of libraries but also of modular software, the linker, and library design. We have included a substantial amount of advice distilled from experience (ours and our students') and from online discussion communities. We found that this helped our students to cope with most of the problems they encountered in courses based on early versions of this book.

We used preliminary versions of this book in Suffolk University's OOP/C++ course each semester during the academic years 2003–2004 through 2005–2006, with increasingly promising results and with much valuable feedback from our students. In the earlier version of this course, students typically would write thousands of lines of code for their programming projects. By contrast, with the emphasis now on code reuse and the exploitation of robust tool libraries, student programming projects have fewer lines of student code, but are much more interesting and, we feel, much more valuable learning experiences.

There are many C++ books out there that either teach C++ or teach Qt, but we found that the C++ books use a variety of different programming styles, and they emphasize some topics that we do not use very often with Qt. The Qt books we have seen all assume prior C++ knowledge. This book, by contrast, assumes no C or C++ programming experience, and it covers the language features of C++ that you need to know in order to use Qt 4 classes as early as possible in the examples and assignments. It can be used as a textbook for teaching C++ and design patterns, with an emphasis on open-source code reuse.

As far as we know, there are no university-level C++ textbooks that contain Qt examples and also provide review questions, exercises, solutions, and lecture slides for instructors.

About the Authors

Paul Ezust started teaching mathematics at Suffolk University in 1967. Soon after completing his Ph.D. in mathematics from Tufts University in 1975, he was appointed chair of the Suffolk University Department of Mathematics, a position that he still holds. Over the next few years he led a successful departmental effort to develop courses and curricula in computer science based on guidelines published by the Association for Computing Machinery (ACM). In 1982 the department was renamed Mathematics and Computer Science. Today the department has very successful undergraduate and graduate programs in computer science and is continuing to grow and develop. Paul has taught computer science courses for nearly 30 years, focusing for the past ten years on object-oriented programming. Over the years he has also done extensive outside consulting, contract programming, and research in computational mathematics. This book, which was originally going to be an extrapolation of a course that he had developed and refined for about eight years, has evolved into one that represents a complete paradigm shift for him and a totally different approach to teaching object-oriented programming, thanks to gentle but persistent pressure from his son Alan.

If you ignore shovelling snow, **Alan Ezust's** first paying job was as a teaching assistant when he was 13. The class was an introduction to Logo on the Apple II, and the students were primary and secondary school teachers, many of whom were using a computer for the first time. Computer language training has been one of his passions ever since. Alan studied computer science at McGill University, receiving his M.Sc. in 1995. He gained more than ten years experience writing course material and teaching programming languages at McGill University, Suffolk University, Learnix Limited, Nortel, Objectivity, Corel, and Hewlett

Packard. While at Learnix, he had the unique experience of getting paid to teach his father C++ in a classroom setting. Little did he know at the time, this would be the start of a long-term professional collaboration. Alan now lives in Victoria, British Columbia. He is working toward a Ph.D. and contributing to the development of open-source software (jEdit, Umbrello, KDE) in his spare time.

An Introduction to Design Patterns in C++ with Qt 4



P A R T I

Introduction to C++ and Qt 4



Chapter 1. C++ Introduction	5
Chapter 2. Classes	47
Chapter 3. Introduction to Qt	81
Chapter 4. Lists	95
Chapter 5. Functions	105
Chapter 6. Inheritance and Polymorphism	135

1

CHAPTER 1

C++ Introduction

In this chapter the language is introduced. Basic concepts such as keywords, literals, identifiers, declarations, native types, and type conversions are defined. Some history and evolution are discussed, along with the relationship between C++ and the C language.

1.1	Overview of C++	6
1.2	A Brief History of C++	6
1.3	Setup: Open-Source Platforms.	7
1.4	Setup: Win32	12
1.5	C++ First Example	12
1.6	Input and Output	16
1.7	Identifiers, Types, and Literals	19
1.8	C++ Simple Types	22
1.9	C++ Standard Library Strings	30
1.10	Streams	31
1.11	The Keyword const	34
1.12	Pointers and Memory Access	36
1.13	const* and *const	40
1.14	Reference Variables	43



1.1 Overview of C++

C++ was originally an extension to C, also known as “C with Objects.” It enhances C by adding several higher-level features such as strong typing, data abstraction, references, operator and function overloading, and considerable support for object-oriented programming.

C++ retains the key features that have made C such a popular and successful language: speed, efficiency, and a wide range of expressiveness that allows programming at many levels, from the lowest (such as direct operating system calls or bitwise operations) to the highest level (manipulating large complex objects or graphs of objects).

A fundamental design decision was made at the beginning for C++: Any features added to C++ should not cause a run-time penalty on C code that does not use them.¹ Certainly, there are added burdens on the compiler, and some features have a run-time cost if they are used, but a C program that is compiled by a C++ compiler should run just as fast as it would if compiled by a C compiler.

1.2 A Brief History of C++

C++ was designed by Bjarne Stroustrup while he was working for AT&T Bell Labs, which eventually packaged and marketed it. Initial versions of the language were made available internally at AT&T beginning in 1981. C++ steadily evolved in response to user feedback.

The first edition of Stroustrup’s book, *The C++ Programming Language*, was published in early 1986. In 1989 with the release of Version 2.0, C++ was rapidly acknowledged as a serious, useful language. In that year, work began on establishing an internationally recognized language standard for C++.

Standardization under the control of ANSI (American National Standards Institute) Committee X3J16 began in 1989 and was completed and published internally in 1997 by the X3 Secretariat under the title *Draft Standard—The C++ Language, X3J16/97-14882, Information Technology Council (NSITC)*,

¹Unfortunately, exception handling broke this rule and does cause a bit of overhead if enabled. This is why many libraries still do not use exceptions.

Washington, DC. In June 1998, the draft standard was unanimously accepted by the representatives of the 20 principal nations that participated in the nine-year ANSI/ISO (International Standards Organization) effort.

The third edition of Stroustrup's book [Stroustrup97], was published in 1997. It is widely regarded as the definitive C++ reference.

You can view an HTML version of the ANSI/ISO Draft Standard at <http://oop.mcs.suffolk.edu/draftstandard>

1.3 Setup: Open-Source Platforms

Open-source development tools (`ssh`, `bash`, `gcc`) are available natively on UNIX and their related platforms. This includes Mac OS/X, which is a BSD-based distribution.²

1.3.1 *nix

When we discuss something that's specific to UNIX-derived platforms (Linux, BSD, Solaris, etc.), we will use the shorthand ***nix** for "most flavors of UNIX."

Another important acronym is **POSIX**, which stands for *Portable Operating System Interface for UNIX*. The development of this family of standards was sponsored by the IEEE (Institute of Electrical and Electronics Engineers), an organization for engineers, scientists, and students that is best known for developing standards for the computer and electronics industry.³

This section is for readers who are using a computer with some flavor of *nix installed.

The examples in this book were tested with Qt 4.1. We recommend that you use the same version or a later one. The first step in setting up your computer for this book is to make sure that the full installation of Qt 4 is available to you. This includes, in addition to the source and compiled library code, the Qt Assistant documentation system, program examples, and the Qt Designer program.



If your system has KDE 3.x (the K Desktop Environment) installed, then there is already a run-time version of Qt 3.x installed. Qt 4 needs to be installed separately. Our examples do not compile under Qt 3.

² BSD stands for Berkeley Software Distribution, a facility of the Computer Systems Research Group (CSRG) of the University of California at Berkeley. CSRG has been an important center of development for UNIX and for various basic protocols (e.g., TCP/IP) since the 1980s.

³ If we wanted to write a POSIX regular expression (see Section 13.2) for *nix, it might look like this: `(lin|mac-os|un|solar|ir|ultr|ai|hp)[iu]?[sx]`.

To see which (if any) version of Qt has already been installed on your system, start with the following commands:

```
which qmake
qmake -v
```

The output of the first command tells you where the qmake executable is located. If that output looks like this: `bash: qmake: command not found`, it is possible that

1. The “full” Qt (including development tools) is not installed
2. It is installed, but your PATH does not include `/path/to/qt4/bin`

If you can run it, `qmake -v` provides version information. If it reports

```
Using Qt version 4.x.y
```

then, check whether these other Qt tools are available:

```
which moc
which uic
which assistant
which designer
```

If these executables are all found, and from the same place as `qmake`, Qt 4 is installed and ready to use.

If the tests outlined above indicate that you have an earlier version or no Qt installed, or that you are missing some components of Qt 4, then you will need to install the latest release of Qt 4.

Installing Qt 4 from Packages

On some *nix systems it is possible to install Qt 4 from a package that contains compiled code.

Using your *nix package manager (e.g., `apt`, `urpmi`, `aptitude`, `kpackage`, `synaptic`, `rpm-drake`, etc.), you can easily and quickly install the packages that comprise Qt 4. Here is a list of packages available on Debian systems (the lists will differ on other distros).

```
[ROOT@lazarus]# apt-cache search qt4
libqt4-core - Qt 4 core non-GUI functionality run-time library
libqt4-debug - Qt 4 debugging run-time libraries
libqt4-dev - Qt 4 development files
libqt4-gui - Qt 4 core GUI functionality run-time library
```

```

libqt4-qt3support - Qt 3 compatibility library for Qt 4
libqt4-sql - Qt 4 SQL database module
qt4-designer - Qt 4 Designer
qt4-dev-tools - Qt 4 development tools
qt4-doc - Qt 4 API documentation
libqt4-designer - Qt 4 Designer libraries
[ROOT@lazarus]#

```

As you can see, in Debian, Qt 4 has been split into many separate packages to give package maintainers more flexibility when they deploy. When developing, you need the full Qt 4 package with developers' tools, full headers, docs, designer, assistant, and examples.

1.3.2 Downloading from Source

You can download, unpack, and compile the latest open-source tarball from Trolltech.⁴ Two typical places to unpack the tarball are `/usr/local/` (if you are root) or `$HOME/qt` (if you are not).



A **tarball** is a file produced by the **tar** command (tape archive) that can combine many files, as specified in the command line, into one file (which is generally given the extension `.tar`) for ease of storage or transfer. The combined file is generally compressed using a utility like `gzip` or `bzip2`, which appends the extension `.gz` or `.bz` to the tar file.

The command line for unpacking a tarball depends on how it was created. You can usually determine this from its extension.

```

tar -vxf whatever.tar // uses the "verbose" switch
tar -zxf whatever.tar.gz // compressed with gzip
tar -zxf whatever.tgz // also compressed with gzip
tar -jxf whatever.tar.bz2 // compressed with bzip2

```

A tar file can preserve directory structures and has many options and switches. You can read the online documentation for these utilities by typing:

```

info tar
info gzip
info bzip

```

⁴ <http://www.troltech.com/download/opensource.html>

The Qt source tarball contains the complete source code of the Qt library, plus numerous examples, tutorials, and extensions with full reference documentation. The tarball contains simple installation instructions (in the `README` and `INSTALL` files) and a `configure --help` message. Be sure to read the `README` file before you attempt to install software from any open-source tarball.

Compiling from source can take 2 to 3 hours (depending on the speed of your system), but it is worth the time. Example 1.1 shows the options we use to configure qt:4.

EXAMPLE 1.1 `./bin/qtconfigure`

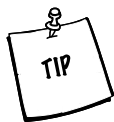
```
#!/bin/sh
# This is a script which I use to configure and
# make qt4 - it includes extra options I use for working with mysql.
#
# Before you run this, be sure you have at least the
# mysql client header files :

# apt-get install libmysqlclient-dev mysql-server mysql-client

INSTALLDIR=/usr/local/qt4
./configure -prefix $INSTALLDIR -fast -qt-gif \
#           for mysql:
#           -I/usr/include/mysql -qt-sql-mysql
#           for odbc:
#           -plugin-sql-odbc -qt-sql-odbc
make
```

You can run this as a regular user as long as you have write permissions in that directory. Notice that this script runs `make` right after it is finished with `configure`, so it will run for quite a while.

In the final step, `make install` copies the executables and headers into another location from the unzipped tarball source tree. If you are installing in a common location, you need to be root to do this.

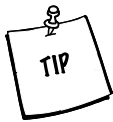


After installation, try `qmake -v` to determine which version of `qmake` is found by your shell. For systems that have more than one version installed, this is especially important to do.

```
> which qmake
/usr/local/qt-x11-opensource-src-4.1/bin/qmake
> qmake -v
QMake version: 2.00a
Using Qt version 4.1 in /usr/local/qt-x11-opensource-
src-4.1/lib
>
```

After installing, check your environment variables and make sure that

- **QTDIR** contains the path to the main directory of the Qt 4 installation.⁵
- **PATH** contains `$QTDIR/bin` (the path to the Qt executables).
- **MANPATH** contains `$QTDIR/doc` (the path to the Qt documentation).
- **LD_LIBRARY_PATH** contains `$QTDIR/lib` (the path to the Qt libraries).



The bash command `env` will display the current values of all your environment variables.

The shell script in Example 1.2 shows how we set the values with `bash`, but the actual values depend on where the files are located on your system.

EXAMPLE 1.2 `src/libs/utills/qt.sh`

```
# Typical user specific environment and startup values for QT
# depending on how and where qt was installed on your system
export QTDIR=/usr/local/qt4
export PATH=$QTDIR/bin:$PATH
export MANPATH=$QTDIR/doc/man:$MANPATH

# Location of your own libraries - source and target.
export CPPLIBS=~/projects/libs

# LD Library path - where to search for shared object libraries
# at runtime.
export LD_LIBRARY_PATH=$CPPLIBS:$QTDIR/lib:$LD_LIBRARY_PATH
```

⁵ Qt 4 does not require the use of this environment variable, but we use this variable to refer to the “Qt install” directory in our examples.

1.4 Setup: Win32

There are two versions of Qt available on Win32 platforms.

1. The open-source edition, which supports MinGW⁶ (the *Minimalist Gnu for Windows*), can be downloaded and used for free with open-source projects.
2. The Qt 4 Commercial Edition can be used with Microsoft compilers, such as Microsoft Visual C++ 6, and later versions of Developer's Studio.

Installing either edition is a snap: The Win32 installer guides you through the process, registers extensions, and sets environment variables for you. The open-source edition can even download and install MinGW for you.

After Qt is installed, you should click

```
Start -> Programs -> Qt by Trolltech -> Build debug symbols
```

This may take a couple of hours.

Next, open a shell window by clicking

```
Start -> Programs -> Qt by Trolltech -> Command Prompt
```

Now you can run `qmake -v` from the command prompt to see the currently installed version of Qt. `qmake`, `assistant`, `designer`, `qtdemo`, `qtconfig`, `g++`, and `make` should all be findable in your search path now.

Try the `qtdemo` that is also available from the Start menu.



MSYS (from MinGW) and Cygwin⁷ both offer `bash` and `xterm`-like shell windows to simulate a *nix system in Win32 environments.

For building the Qt debug symbols, we found that the `configure` and `make` scripts did not work. However, for building our own apps, we have been able to run `qmake` and `make` inside a `bash` shell from Cygwin.

1.5 C++ First Example

Throughout this book we will use code examples to explain and illustrate important programming and OOP issues. Our aim in each case is to use a minimal example that will illustrate the ideas and techniques briefly and efficiently. Example 1.3 provides a quick overview of some elementary C++ syntax.

⁶ <http://www.mingw.org/>

⁷ <http://www.cygwin.com>

EXAMPLE 1.3 `src/early-examples/fac.cpp`

```

// Computes and prints n! for a given n.
// Uses several basic elements of C++.

#include <iostream>                                ❶

int main() {                                       ❷
    using namespace std;                          ❸
    /* Declarations of variables */
    int factArg = 0 ;                             ❹
    int fact (1) ;                                ❺
    do {                                           ❻
        cout << "Factorial of: ";                ❼
        cin >> factArg;                          ❽
        if( factArg < 0 ) {                       ❾
            cout << "No negative values, please!" << endl;
        }
    } while(factArg < 0) ;                         ❿
    int i = 2;
    while( i <= factArg ) {                       ⓫
        fact = fact * i;
        i = i + 1;
    }                                              ⓬
    cout << "The Factorial of " << factArg << " is: " << fact << endl;
    return 0;                                     ⓭
}                                                  ⓮

```

- ❶ Standard c++ library. In older versions of C++, you might find `<iostream.h>` instead, but that version is regarded as obsolete or "deprecated."
- ❷ start of function "main," which returns int
- ❸ Permits us to use the symbols `cin`, `cout`, and `endl` without prefixing each name with `std::`
- ❹ C style initialization syntax
- ❺ C++ style initialization syntax
- ❻ start of "do..while" loop
- ❼ write to standard output
- ❽ read from standard input and convert to int
- ❾ end of if block
- ❿ if false, break out of do loop
- ⓫ start of while loop
- ⓬ end of while block
- ⓭ When main returns 0, that normally means "no errors."
- ⓮ end of main block

On most platforms it is simple to compile and run this program using the ubiquitous GNU C compiler, `gcc`. The command to compile a C++ program is `g++`, but this is simply an alias to `gcc` with some C++ options switched on.

When invoking the compiler, we recommend always maximizing the amount of diagnostic information that is available from the compilation process. Accordingly, we include three switches on the command line: (1) `-ansi` (which turns off GNU

extensions that conflict with ISO C++), (2) `-pedantic` (which issues all the warnings that are demanded by strict ISO C++ and rejects any program that uses forbidden extensions), and (3) `-Wall` (which enables all possible warnings about constructions that might be considered questionable even if they conform to the standard).

```
src/early-examples> g++ -ansi -pedantic -Wall fac.cpp
```

or

```
src/early-examples> g++ -ansi -pedantic -Wall -o execFile fac.cpp
```

In the second version shown above, `-o execFile` is an optional switch that tells the compiler to name the executable result of the compilation “*execFile*.” If we omit that switch, as in the first version, the compiler will produce an executable file named `a.out`. In either case, if there already exists a file in the same directory with the name of our target executable, then the compiler will quietly and automatically overwrite it.

We have mentioned here just a few of the most commonly used compiler switches. On a *nix system you can view the **manual pages**, a summary of the `g++` command options and how they are used, by typing the command

```
man g++
```

The command

```
info g++
```

often displays more readable documentation. One advantage is that the `info` command allows you to do an incremental search for a word in the documentation by typing `ctrl-s`. On most systems these commands allow you to browse the online documentation for `g++` one screen at a time. For more complete `gcc` documentation, visit the GNU online document center.⁸

After it has been compiled successfully, our program can be run by typing the name of the executable file. Here is an example done on a *nix platform:

```
src/early-examples> ./a.out
Factorial of: -3
No negative values, please!
Factorial of: 5
The Factorial of 5 is: 120
src/early-examples>
```

⁸ <http://www.gnu.org/software/gcc/onlinedocs/>

This short program has several of the language elements that show up in most C++ programs. Some interesting differences between C++ and other languages (especially C) can be seen in this example.

Comments

C++ has single-line comments as in Java. Any text between the `//` and the end of the line is a comment. C-style comment delimiters for multiline comments can also be used: The text between `/*` and `*/` is a comment.

#include

To reuse functions, types, or identifiers from libraries, we use the preprocessor directive `#include` (see Section C.1). As in C, all preprocessor directives begin with the pound sign character, `#`, and are evaluated just before the compiler compiles your code. In this example, the included header `<iostream>` contains the Standard Library definitions for input/output.

The Using Namespace Directive

Symbols from the Standard Library (see Appendix B) are all enclosed in the namespace `std`.

A **namespace** (see Section 20.4) is a collection of classes, functions, and objects that can be addressed with a name prefix. The **using declaration** tells the compiler to add all symbols from the namespace `std` to our global namespace.

Declaring and Initializing Variables

Variable declarations come in three styles in C++:

```
type-expr  variableName;
type-expr  variableName = init-expr;
type-expr  variableName (init-expr);
```

In the first form, the variable might not be initialized. The third form is an alternate syntax for the second.

Selection

C++ provides the usual assortment of syntax variations for selection, which we discuss in Section 21.2.

Iteration

We used two of the three iteration structures provided by C++ in our example. All three are discussed in Section 21.3.

1.6 Input and Output

In Example 1.3, the directive

```
#include <iostream>
```

allowed us to make use of the predefined global input and output stream objects. They are

1. `cin`—**standard input**, the keyboard by default
2. `cout`—**standard output**, the console screen by default
3. `cerr`—**standard error**, another output stream to the console screen that flushes more often and is normally used for error messages

To send output to the screen in Example 1.3, we made use of the global stream object (of the class `ostream`), called `cout`. We called one of its member functions, whose name is `operator<<()`. This function overloads the insertion operator `<<` and defines it like a global function.⁹ The syntax for that output statement is also quite interesting. Instead of using the rather bulky function notation:

```
cout.operator<<("Factorial of :");
```

we invoked the same function using the more elegant and readable *infix* syntax:

```
cout << "Factorial of:  " ;
```

This operator is predefined for use with many built-in types, as we see in the next output statement.

```
cout << "The cost is $" << 23.45 << " for " << 6 <<
"items." << '\n';
```

In Example 1.4, we can see the `operator>>()` used for input with `istream` in an analogous way to `<<` for `ostream`.

⁹ We will discuss overloaded functions and operators in more detail later in Section 5.2.

EXAMPLE 1.4 `src/iostream/io.cpp`

```
#include <string>
#include <iostream>

int main() {
    using namespace std;
    const int THISYEAR = 2006;
    string yourName;
    int birthYear;

    cout << " What is your name? " << flush;
    cin >> yourName;

    cout << "What year were you born? " ;
    cin >> birthYear;

    cout << "Your name is " << yourName
         << " and you are approximately "
         << (THISYEAR - birthYear)
         << " years old. " << endl;
}
```

The symbols `flush` and `endl` are **manipulators** that were added to the `std` namespace for convenience.¹⁰ Manipulators are implicit calls to functions that can change the state of a stream object in various ways.

Notice also that we are using the `string` type from the Standard Library. We will discuss this type and demonstrate some of its functions shortly (see Section 1.9).

EXERCISES: INPUT AND OUTPUT

1. Using Example 1.4, do the following experiments:
 - First, compile and run it to see its normal behavior.
 - What happens if you enter a non-numeric value for the birth year?
 - What happens if you enter a name such as Curious George as your name?
 - What happens if you remove the following line?
`using namespace std;`

¹⁰ We discuss these further in Section 1.10.

- Replace the statement
`cin >> yourName;`
with the statement
`getline(cin, yourName);`
and try Curious George again.
- Can you explain the differences in behavior between `cin >>` and `getline()`? We discuss this later in Section 1.9.
- Add some more questions to the program that require a variety of numerical and string answers, and test the results.

2. Consider the program shown in Example 1.5.

EXAMPLE 1.5 `src/early-examples/fac2.cpp`

```
#include <iostream>

long factorial(long n) {
    long ans = 1;
    for (long i = 2; i <= n; ++i) {
        ans = ans * i;
        if (ans < 0) {
            return -1;
        }
    }
    return ans;
}

int main() {
    using namespace std;
    cout << "Please enter n: " << flush;
    long n;           ❶
    cin >> n;         ❷

    if (n >= 0) {
        long nfact = factorial(n);
        if (nfact < 0) {
            cerr << "overflow error: "
                 << n << " is too big." << endl;
        }
    }
}
```

```

        else {
            cout << "factorial(" << n << ") = "
                 << nfact << endl;
        }
    }
    else {
        cerr << "Undefined:  "
             << "factorial of a negative number: " << n << endl;
    }
    return 0;
}

```

- ❶ long int
- ❷ read from stdin, try to convert to long

- Test the program in Example 1.5 with a variety of input values, including some non-numeric values.
- Determine the largest input value that will produce a valid output.
- Can you change the program so that it will produce valid results for larger input values?
- Modify the program so that it cannot produce invalid results.
- Explore the effects of using the statement

```
if(cin >> n)    { ... }
```

to enclose the processing of `n` in this program. In particular, try entering non-numeric data after the prompt. This is an example of the use of a conversion operator, which we discuss in more detail in Section 2.13.

- Modify the program so that it will accept values from the user until the value 9999 is entered.

1.7 Identifiers, Types, and Literals

Identifiers are names that are used in C++ programs for functions, parameters, variables, constants, classes, and types. An identifier consists of a sequence of letters, digits, and underscores that does not begin with a digit. An identifier cannot be a reserved keyword. See Appendix A for a list of them. The standard does not specify a limit to the length of an identifier, but certain implementations of C++ only examine the first 31 characters to distinguish between identifiers.

A **literal** is a constant value that appears somewhere in a program. Since every value has a type, every literal has a type also. It is possible to have literals of each of the native data types, as well as character string literals. Table 1.1 shows some examples of literals.

TABLE 1.1 Examples of Literals

Literal	Meaning
5	an <code>int</code> literal
5u	u or U specifies unsigned <code>int</code>
5L	l or L specifies long <code>int</code> after an integer
05	an octal <code>int</code> literal
0x5	a hexadecimal <code>int</code> literal
true	a <code>bool</code> literal
5.0F	f or F specifies single precision floating point literal
5.0	a double precision floating point literal
5.0L	l or L specifies long double if it comes after a floating point
'5'	a <code>char</code> literal (ASCII 53)
"50"	a <code>const char*</code> containing the chars '5', '0', and '\0'
"any" "body"	"anybody"
'\a'	alert
'\\'	backslash
'\b'	backspace
'\r'	carriage return
'\''	single quote
'\"'	double quote
'\f'	formfeed (newpage)
'\t'	tab
'\n'	newline <code>char</code> literal
"\n"	newline followed by null terminator (<code>const char*</code>)
'\0'	null character
'\v'	vertical tab
"a string with newline\n"	another <code>const char*</code>

String literals are special in C++, due to its historical roots in the C language. Example 1.6 shows how certain characters need to be escaped inside double-quoted string delimiters.

EXAMPLE 1.6 `src/early-examples/literals.cpp`

```
#include <iostream>
#include <string>

int main() {
    using namespace std;
    const char* charstr = "this is one very long string "
                          "so I will continue it on the next line";
    string str = charstr; ❶
    cout << str << endl;
    cout << "\nA\tb\\c'd\" << endl;
    return 0;
}
```

❶ STL strings can hold onto C-style `char*` strings.

We compile and run the way we described earlier:

```
src/early-examples> g++ -ansi -pedantic -Wall literals.cpp
src/early-examples> ./a.out
```

The output should look something like this:

```
this is one very long string so I will continue it on the next line
A      b\c'd"
```

Notice that this program shows a way to avoid very long lines when dealing with string literals. They can be broken at any white space character and are concatenated automatically using this syntax.

EXERCISES: IDENTIFIERS, TYPES, AND LITERALS

Modify Example 1.6 so that, with a single output statement, the output becomes:

1. GNU stands for "GNU's Not Unix".
2. Title 1 "Cat Clothing"
 Title 2 "Dog Dancing"

1.8 C++ Simple Types

The simple types supported in C/C++ are

- A Boolean type (`bool`)
- Character types (`char` and `wchar_t`)
- Integer types (`short`, `int`, `long`)
- Floating point numbers (`double`, `float`, `long double`, etc.)
- Pointers (`int*`, `char*`, `bool*`, `double*`, `void*`, etc.)

In addition, C and C++ both provide a name that signifies the absence of type information (`void`).

C++ simple types can (variously) be modified by the following keywords to produce other simple types:

- `short`
- `long`
- `signed`
- `unsigned`¹¹

TABLE 1.2 Simple Types Hierarchy

Byte/char types	Integral types	Floating point types
<code>bool</code>	<code>short int</code>	<code>float</code>
<code>char</code>	<code>int</code>	<code>double</code>
<code>signed char</code>	<code>long int</code>	<code>long double</code>
<code>unsigned char</code>	<code>unsigned short</code>	
<code>wchar_t</code>	<code>unsigned int</code>	
	<code>unsigned long</code>	

C++ compilers allow you to omit “`int`” from `short int`, `long int`, and `unsigned int`. You can omit `signed` from most types, since that is the default.

Since the range of values for a particular type depends on the underlying architecture of the machine on which the compiler is running, the ANSI/ISO standard for C++ does not specify the size (in bytes) of any of these types. It only guarantees

¹¹ For a brief discussion of the differences between signed and unsigned integral types, see Section 19.4.

that a given type (e.g., `int`) must not be smaller than one that appears above it (e.g., `short`) in Table 1.2.

There is a special operator `sizeof()` that returns the number of `chars`¹² that a given expression requires for storage. Unlike most functions, the `sizeof()` operator can take value expressions or type expressions.

Example 1.7 shows how `sizeof()` can be used, and some of the values it returns on a 32-bit x86 system.

EXAMPLE 1.7 `src/early-examples/size.cpp`

```
#include <assert.h>
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    using namespace std;
    int i=0;
    char array1[34] = "This is a dreaded C array of char";
    char array2[] = "if not for main, we could avoid it entirely.";
    char *charp = array1; ❶
    string stlstring = "This is an Standard Library string. Much
preferred." ;
    assert (sizeof(i) == sizeof(int));
    cout << "sizeof char = " << sizeof(char) << '\n';
    cout << "sizeof wchar_t = " << sizeof(wchar_t) << '\n';
    cout << "sizeof int = " << sizeof(int) << '\n';
    cout << "sizeof long = " << sizeof(long) << '\n';
    cout << "sizeof float = " << sizeof(float) << '\n';
    cout << "sizeof double = " << sizeof(double) << '\n';
    cout << "sizeof double* = " << sizeof(double*) << '\n';
    cout << "sizeof array1 = " << sizeof(array1) << '\n';
    cout << "sizeof array2 = " << sizeof(array2) << '\n';
    cout << "sizeof stlstring = " << sizeof(stlstring) << endl;
    cout << "length of stlstring= " << stlstring.length() << endl;
    cout << "sizeof char* = " << sizeof(charp) << endl;
    return 0;
}
```

Output:

```
sizeof char = 1
sizeof wchar_t = 4
sizeof int = 4
sizeof long = 4
```

continued

¹² On most systems, a single `char` is a byte.

```
sizeof float = 4
sizeof double = 8
sizeof double* = 4
sizeof array1 = 34
sizeof array2 = 46
sizeof stlstring = 4
length of stlstring = 38
sizeof char* = 4
```

① pointer to first element of array

Notice that all pointers are the same size, regardless of their type. `sizeof(stlstring)` indicates it is only 4 bytes, but it is a complex class that uses dynamic memory, so we use `length()` to get the number of characters in the string.

The ranges of values for the integral types (`bool`, `char`, `int`) are defined in the standard header file `limits.h`. On a typical *nix installation that file can be found in a subdirectory of `/usr/include`.

1.8.1 main and Command Line Arguments

`main()` is a function (see Chapter 5), that is called at program startup. If the program accepts command line arguments, we must define `main` with its full parameter list.

C permits flexibility in the ways that arguments are defined in `main()`, so you may see it defined in a variety of ways:

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

All of the above forms are valid, and each defines two parameters. These parameters contain enough information to reconstruct the command line arguments passed into the program from the parent process (a command line shell, a window manager, etc.).

Example 1.8 is a simple `main` program that prints its command line arguments.

EXAMPLE 1.8 `src/main/main1.cpp`

```
#include <iostream>
using namespace std;
```

```
int main (int argCount, char* argValue[]) {
    for (int i=0; i<argCount; ++i) {
        cout << "argv# " << i << " is " << argValue[i] << endl;
    }
    return 0;
}
```

`argValue`, or `argv` for short, is a two-dimensional array (see Section 22.4) of `char`. `argCount`, or `argc` for short, is the number of `char` arrays in `argv`. `argv` contains each command line string as an item in its array.

`int main()` “returns” an integer, which should be 0 if all went well, or a non-zero error code if something went wrong.



Try not to confuse this interpretation of 0 with the `bool` value `false`, which is equal to zero.

If we run this program and pass some arguments, we will see something like this in the output:

```
~/src/main> ./a.out foo bar "space wars" 123
argv# 0 is ./a.out
argv# 1 is foo
argv# 2 is bar
argv# 3 is space wars
argv# 4 is 123
```

The first argument is always the name of the executable. The other arguments are taken from the command line as strings separated by spaces or tabs. To pass a string that contains spaces as a single argument, you must enclose the string in quotes.

1.8.2 Arithmetic

Each programming language must provide facilities for doing basic arithmetic. For each of its native numerical types, C++ provides these four basic arithmetic operators:

- Addition (+)
- Subtraction (-)

- Multiplication (*)
- Division (/)

These operator symbols are used to form expressions in the standard infix syntax that we learned in math class.

C++ provides shortcut operators that combine each of the basic operators with the assignment operator (=) so that, for example, it is possible to write

```
x += y;
```

instead of

```
x = x + y;
```

C++ also provides unary increment (++) and decrement (--) operators that can be used with integral types. If one of these operators is applied to a variable on the left (prefix), then the operation is performed before the rest of the expression is evaluated. If it is applied to a variable on the right (postfix), then the operation is performed after the rest of the expression is evaluated. Examples 1.9 through 1.13 demonstrate the use of the C++ arithmetic operators.

EXAMPLE 1.9 `src/arithmetic/arithmetic.cpp`

```
[ . . . . ]
#include <iostream>

int main() {
    using namespace std;
    double x(1.23), y(4.56), z(7.89) ;
    int i(2), j(5), k(7);
    x += y ;
    z *= x ;
    cout << "x = " << x << "\tz = " << z
         << "\nx - z = " << x - z << endl ;
}
```

Integer division is handled as a special case. The result of dividing one `int` by another produces an `int` quotient and an `int` remainder. The operator `/` is used to obtain the quotient. The operator `%` (called the **modulus operator**) is used to obtain the remainder.

Example 1.10 shows the use of these integer arithmetic operators.

EXAMPLE 1.10 `src/arithmatic/arithmatic.cpp`

[. . . .]

```

cout << "k / i = " << k / i
      << "\tk % j = " << k % j << endl ;
cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;
cout << "++k / i = " << ++k / i << endl;
cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;
cout << "i * j-- = " << i * j-- << endl;
cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;

```

Mixed expressions, if valid, generally produce results that are of the widest of the argument types. We discuss this in more detail in Chapter 19.

Example 1.11 shows that the result of a `double` divided by an `int` is a `double`.

EXAMPLE 1.11 `src/arithmatic/arithmatic.cpp`

[. . . .]

```

cout << "z / j = " << z / j << endl ;

```

C++ also provides a full set of boolean operators to compare numeric expressions. Each of these operators returns a `bool` value of either `false` or `true`.

- Less than (`<`)
- Less than or equal to (`<=`)
- Equal to (`==`)
- Not equal to (`!=`)
- Greater than (`>`)
- Greater than or equal to (`>=`)

A `bool` expression can be negated with the unary `not` (`!`) operator. Two `bool` expressions can be combined with the operators

- `and` (`&&`)
- `or` (`||`)

We discuss `bool` expressions in more detail in Section 19.2.

EXAMPLE 1.12 `src/arithmetic/arithmetic.cpp`

```
[ . . . . ]

/*  if() ... else  approach */
if(x * j <= z)
    cout << x * j << " <= " << z << endl ;
else
    cout << x * j << " > " << z << endl;
/* conditional operator approach */
cout << x * k
      <<( (x * k < y * j) ? " < " : " >= ")
      << y * j << endl;
}
```

In addition to the binary boolean operators, Example 1.12 makes use of the *conditional-expression*.

The expression

```
(boolExpr) ? expr1 : expr2
```

returns *expr1* if *boolExpr* is `true`, otherwise it returns *expr2*. Example 1.13 shows the output that this program produces.

EXAMPLE 1.13 `src/arithmetic/arithmetic.cpp`

```
[ . . . . ]
```

Output:

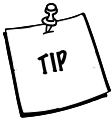
```
x = 5.79          z = 45.6831
x - z = -39.8931
k / i = 3        k % j = 2
i = 2   j = 5   k = 7
++k / i = 4
i = 2   j = 5   k = 8
i * j-- = 10
i = 2   j = 4   k = 8
z / j = 11.4208
23.16 <= 45.6831
46.32 >= 18.24
```

EXERCISES: C++ SIMPLE TYPES

1. Here is an old favorite: Write a short program that asks the user to choose Celsius-to-Fahrenheit or Fahrenheit-to-Celsius. Then ask for a lower bound, an upper bound, and an increment. Using that information, produce an appropriate table with column headings. The `main()` part of your program should be relatively short. Most of the work should be performed by functions that you design, e.g., `celsiusToFahrenheit()`, `fahrenheitToCelsius()`, `makeTable()`, and so forth (with appropriate parameter lists).
2. If you `#include <cstdlib>`, you can make use of the `random()` function that generates a sequence of pseudo-random long ints in the range from 0 to `RAND_MAX`, which can be used in many interesting ways. It works by computing the next number in its sequence from the last number that it generated. The function `srandom(unsigned int seed)` sets its argument as the initializing value (seed) for the random sequence. Write a short program that tests this function by allowing the user to supply the seed from the keyboard and then generating a list of pseudo-random numbers.
3. One trick is to use `srandom(time(0))` to seed the `random()` function. Since `time(0)` returns the number of seconds since some initial starting point, the seed will be different each time you run the program. This allows you to write programs that have usefully unpredictable behavior patterns. Write a program that simulates a dice game that the user can play with the computer. Here are the rules to apply to your game:
 - The game is about repeated “throws” of a pair of dice.
 - Each die has six faces, which are numbered 1 through 6.
 - A throw results in a number that is the total of the two top faces.
 - The first throw establishes the player’s number.
 - If that number is 7 or 11, the player automatically wins.
 - If that number is 2, the player automatically loses.
 - Otherwise, the player continues throwing until she wins (by matching her number) or loses (by throwing a 7 or an 11).
4. Write a program that accepts two values from the user (customer): the total purchase amount and the amount submitted for payment.

Each of these values can be stored in a variable of type `double`. Then compute and display the change that will be given to the user. Express the change in terms of the number of \$10 bills, \$5 bills, \$1 bills, quarters, dimes, nickels, and pennies. (Presumably, this output could be sent to a machine that dispenses those items automatically.)

For example, if the total purchase amount is \$73.82 and the customer pays with a \$100 bill, then the change should be: two \$10 bills, one \$5 bill, one \$1 bill, no quarters, one dime, one nickel, and three pennies.



Convert the amount that is owed the customer to pennies, which can be stored as an `int` and then use the integer division operators.

1.9 C++ Standard Library Strings

Our early (pre-Qt) examples will make use of C++ Standard Library strings (see Appendix B). Standard Library strings have many disadvantages when compared to QStrings, but they are easy to use and have a similar public interface that includes many functions to construct and modify strings.

Example 1.14 demonstrates its basic usage.

EXAMPLE 1.14 `src/generic/stlstringdemo.cpp`

```
#include <string>
#include <iostream>

int main() {
    using namespace std;
    string s1("This "), s2("is a "), s3("string.");
    s1 += s2; ❶
    string s4 = s1 + s3;
    cout << s4 << endl;

    string s5("The length of that string is: ");
    cout << s5 << s4.length << " characters." << endl;

    cout << "Enter a sentence: " << endl;
    getline(cin, s2);
    cout << "Here is your sentence: \n" << s2 << endl;
    cout << "The length of it was: " << s2.length() << endl;
    return 0;
}
```

❶ concatenation

Here is the compile and run:

```
src/generic> g++ -ansi -pedantic -Wall stlstringdemo.cpp
src/generic> ./a.out
This is a string.
The length of that string is 17
Enter a sentence:
20 years hard labour
Here is your sentence:
20 years hard labour
The length of it was: 20
src/generic>
```

Observe that we used the `getline(istream, string)` function to take a string from standard input stream. We will learn more about input from streams in the following section.

1.10 Streams

Streams are objects used for reading and writing. The Standard Library defines `<iostream>`, while Qt defines `<QTextStream>` for the equivalent functionality. `iostream` defines the three global streams:

- `cin`—standard input (keyboard)
- `cout`—standard output (console screen)
- `cerr`—standard error (console screen)

Also defined (in both `<iostream>` and `<QTextStream>`) are manipulators, such as `flush` and `endl`. A manipulator can be added to

- An output stream to change the way the output data is formatted
- An input stream to change the way that the input data is interpreted

The code in Example 1.15 demonstrates the use of several manipulators applied to the standard output stream.

EXAMPLE 1.15 `src/stdstreams/streamdemo.cpp`

```
#include <iostream>

int main() {
    using namespace std;
    int num1(1234), num2(2345) ;
```

continued

```

cout << oct << num2 << '\t'
      << hex << num2 << '\t'
      << dec << num2
      << endl;
cout << (num1 < num2) << endl;
cout << boolalpha
      << (num1 < num2)
      << endl;
double dub(1357);
cout << dub << '\t'
      << showpos << dub << '\t'
      << showpoint << dub
      << endl;
dub = 1234.5678;
cout << dub << '\t'
      << fixed << dub << '\t'
      << scientific << dub << '\n'
      << noshowpos << dub
      << endl;
}

```

Output:

```

4451    929    2345
1
true
1357    +1357    +1357.00
+1234.57    +1234.567800    +1.234568e+03
1.234568e+03

```

Streams are used for reading from or writing to files, network connections, and also strings. One useful feature of streams is that they make it easy to produce strings from mixed types of data. In Example 1.16, we will create some strings from numerics and write them to a file.

EXAMPLE 1.16 src/stl/streams/streams.cpp

```

[ . . . . ]

int main() {
    using namespace std;
    ostringstream strbuf;

    int lucky = 7;
    float pi=3.14;
    double e=2.71;

```

```

/* An in-memory stream */
strbuf << "luckynumber " << lucky << endl
        << "pi " << pi << endl
        << "e " << e << endl;

string strval = strbuf.str(); ❶
cout << strval;

/* An output file stream. */
ofstream outf;
outf.open("mydata");        ❷
outf << strval ;
outf.close();

```

-
- ❶ convert the stringstream to a string
 - ❷ creates (or overwrites) a disk file for output
-

After the strings have been written, we have a couple of choices for how to read them. We can use the analogous input operators that we wrote to output, and because there is whitespace between each record, the insertion operator will work as shown in Example 1.17.

EXAMPLE 1.17 `src/stl/streams/streams.cpp`

```

[ . . . . ]

/* An input file stream */
ifstream inf;
inf.open("mydata");
string newstr;
int lucky2;
inf >> newstr >> lucky2;
if (lucky != lucky2)
    cerr << "ERROR! wrong lucky number" << endl;

float pi2;
inf >> newstr >> pi2;
if (pi2 != pi) cerr << "ERROR! Wrong pi." << endl;

double e2;
inf >> newstr >> e2;
if (e2 != e) cerr << "e2: " << e2 << " e: " << e << endl;
inf.close();

```

In addition, we can read files line-by-line and deal with each line as a string, as shown in Example 1.18.

EXAMPLE 1.18 `src/stl/streams/streams.cpp`

```
[ . . . . ]

    /* Read line-by-line */
    inf.open("mydata");
    while(not inf.eof()) {
        getline(inf, newstr);
        cout << newstr << endl;
    }
    inf.close();
    return 0;
}
```

EXERCISE: STREAMS

Modify the program in Example 1.16 so that it does the following:

- It gets the file name from the user as an STL string `fileName`. You will need to use the function `fileName.c_str()` to convert the string to a form that is acceptable to the `open()` function.
- It makes sure that the file specified by the user does not already exist (or that it is all right to overwrite it if it does exist) before opening it for output.
- It makes sure that the file exists before attempting to read from it. (Hint: After the call to `open`, you can test the `ifstream` variable as if it were a `bool`. `false` means that the file does not exist.)

1.11 The Keyword `const`

Declaring an entity to be `const` tells the compiler to make it “read-only.” `const` can be applied usefully in a large number of programming situations, as we will soon see.

Because it cannot be assigned to, a `const` object must be properly initialized. For example:

```
const int x = 33;
const int v[] = {3, 6, x, 2 * x};
```

Working with the declarations above:

```
++x ;           // error
v[2] = 44;      // error
```

Compilers can take advantage of an object being read-only in various ways. For integers and some simple types, no storage needs to be allocated for a `const` unless its address is taken. Therefore, most optimizing compilers try to store them in static memory.

It is good programming style to use `const` entities instead of embedding constant expressions (sometimes called “magic numbers”) in your code. This will gain you flexibility later when you need to change the values and, in general, it will improve the maintainability of your programs. For example, instead of something like this:

```
for(i = 0; i < 327; ++i) {
    ...
}
```

use something like this:

```
// const declaration section of your code
const int SIZE = 327;
...
for(i = 0; i < SIZE; ++i) {
    ...
}
```



In some C/C++ programs, you might see constants defined with preprocessor macros like this:

```
#define STRSIZE 80

[...]

char str[STRSIZE];
```

Preprocessor macros get replaced before the compiler sees them. Using macros instead of constants means that the compiler cannot perform the same level of type checking as it can with proper constant expressions. Generally `const` expressions are preferred to macros for defining constant values in C++ programs.

1.12 Pointers and Memory Access

C and C++ distinguish themselves from many other languages by permitting direct access to memory through the use of pointers. This section explains the basic pointer operations and modifiers, and introduces dynamic memory usage. Pointers can seem complicated at first. We discuss pointer use and misuse in more detail in Chapter 22.

1.12.1 The Unary Operators `&` and `*`

A **variable** is an object with a name recognized by the compiler. A variable's name can be used as if it is the object itself. For example, if we say:

```
int x = 5;
```

we can use `x` to stand for the integer object whose value is 5, and we can manipulate the integer object directly through the name `x`. For example:

```
++x ; // symbol x now refers to an integer with value 6
```

An **object** (in the most general sense) is a chunk of memory that can hold data. Each object has a memory address (where the data begins). The **unary `&` operator**, also known as the **address-of operator**, when applied to any object, returns the memory address of that object. For example, `&x` returns the memory address of `x`.

An object that holds the memory address of another object is called a **pointer**. We say that the pointer **points** to the object at the stored memory address.

```
int* y = &x ;
```

In this example, `y` points to the integer `x`. The asterisk `*` following the `int` indicates that `y` is a **pointer to `int`**. Here we have initialized the `int` pointer `y` to the address of the `int` variable `x`. One of the powerful features of pointers is that, subject to rules that we will explore shortly, it is possible for a pointer of one type to hold the address of an object of a different (but related) type.

Zero (0), often represented by the macro `NULL` in C programs, is a special value that can be legally assigned to a pointer, usually when it is being initialized (or re-initialized). 0 is not the address of any object. A pointer that stores the value 0 is called a **null pointer**. Stroustrup recommends the use of 0 rather than the macro `NULL` in C++ programs.

A pointer to a simple type uses exactly the same amount of memory as a pointer to a large complicated object. That size is usually the same as `sizeof(int)` on that machine.

The **unary * operator**, also known as the **dereference operator**, when applied to a non-null pointer returns the object at the address stored by the pointer.



The symbol * is used in two different ways in connection with pointers:

- It can serve as a *type modifier* in a pointer variable definition.
- It can be used as the dereference operator.



Dereferencing a null or uninitialized pointer causes a run-time error, usually a segmentation fault or, in Windows, a General Protection Fault (GPF).

It is the responsibility of the programmer to make sure that no attempt is made to dereference a null or uninitialized pointer. We will discuss techniques to ensure that such errors are avoided.

EXAMPLE 1.19 src/pointers/pointerdemo.cpp

```
// Filename pointerdemo.cpp

#include <iostream>
using namespace std;

int main() {
    int x = 4;
    int* px = 0 ;           ❶
    px = &x;
    cout << "x = " << x
         << " *px = " << *px  ❷
         << " px = " << px
         << " &px = " << &px << endl;
    x = x + 1;
    cout << "x = " << x
         << " *px = " << *px
         << " px = " << px << endl;
    *px = *px + 1;
    cout << "x = " << x
         << " *px = " << *px
         << " px = " << px << endl;
    return 0;
}
```

continued

Output:

```

OOP> ./a.out
x = 4 *px = 4 px = 0xbffff514 &px = 0xbffff510
x = 5 *px = 5 px = 0xbffff514
x = 6 *px = 6 px = 0xbffff514
OOP>

```

- ❶ type modifier
- ❷ unary dereference operator

The particular values of the addresses will, of course, be different when the code in Example 1.19 is executed on different machines.

The variable `x` accesses its data *directly*, but the variable `px` accesses the same data *indirectly*. This is why the word **indirection** is often used to characterize the process of accessing data through a pointer. The relationship between the two variables, `x` and `px`, is illustrated in Figure 1.1.

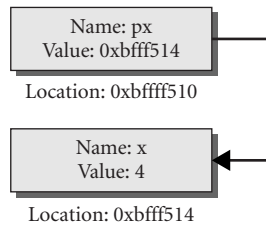


FIGURE 1.1 Pointer demo

1.12.2 Operators `new` and `delete`

C++ has a mechanism that permits storage to be allocated dynamically at runtime. This means that the programmer does not need to anticipate the memory needs of a program in advance and make allowances for the maximum amount of memory that might be needed by the program. Dynamic allocation of storage at runtime is a powerful tool that helps to build programs that are efficient and flexible.

The `new` operator allocates storage from the **memory heap** (also called the heap, free pool, or free storage) and returns a pointer to the newly allocated object. If for some reason it is not possible for the memory to be allocated, an exception is thrown (see Section 22.9).



In general, the code that calls `new` should document, or be physically located near, the code that frees the memory. The goal is to keep memory management code as simple and reliable as possible.

The `delete` operator releases dynamically allocated memory and returns it to the memory heap. `delete` should be applied only to pointers returned by the `new`, or to null pointers. Heap memory that is no longer needed should be released for reuse. Failure to do so can result in crippling memory leaks.



Qt, the Standard Library, and `boost.org` provide a variety of classes and functions to help manage and clean up heap memory. In addition to container classes, each library has one or more smart pointer class. A smart pointer is an object that stores and manages a pointer to a heap object. It behaves much like an ordinary pointer except that it automatically deletes the heap object at the appropriate time. Qt has `QPointer`, the Standard Library has `std::auto_ptr`, and Boost has a `shared_ptr`. Using one of these classes makes C++ memory management much easier than it used to be.

The syntax of the `new` and `delete` operators is demonstrated in the code fragment shown in Example 1.20.

EXAMPLE 1.20 `src/pointers/newdelete/ndysntax.cpp`

```
{
  int* ip = 0;           ❶
  ip = new int;         ❷
  int* jp = new int(13); ❸
  [...]
  delete ip;           ❹
  delete jp;
}
```

- ❶ null pointer
- ❷ allocate space for an int
- ❸ allocate and initialize
- ❹ Without this, we have a memory leak.

EXERCISES: POINTERS AND MEMORY ACCESS

1. Predict the output of the program shown in Example 1.21.

EXAMPLE 1.21 `src/pointers/newdelete1.cpp`

```
#include <iostream>
using namespace std;

int main() {
    const char tab = '\t';
    int n = 13;
    int* ip = new int(n + 3);
    double d = 3.14;
    double* dp = new double(d + 2.3);
    char c = 'K';
    char* cp = new char(c + 5);
    cout << *ip << tab << *dp << tab << *cp << endl;
    int* ip2 = ip;
    cout << ip << tab << ip2 << endl;
    *ip2 += 6;
    cout << *ip << endl;
    delete ip;
    cout << *ip2 << endl;
    cout << ip << tab << ip2 << endl;
}
```

Then compile and run the code. Explain the output, especially the last two lines.

2. Modify Example 1.19 to do some arithmetic with the value pointed to by `jp`. Assign the result to the location in memory pointed to by `ip`, and print the result. Print out the values from different places in the program. Investigate how your compiler and run-time system react to placement of the output statements.

1.13 `const*` and `*const`

Suppose that we have a pointer `ptr` that is storing the address of a variable `vbl`:

```
Type* ptr = &vbl;
```

When using a pointer, two objects are involved: the pointer itself and the object pointed to. This means there are two possible layers of protection that we might want to impose with `const`:

1. If we want to make sure that `ptr` cannot point to any other memory location, we can write it one of two ways:

```
Type* const ptr = &vbl;
Type* const ptr(&vbl);
```

The pointer is a `const` but the addressed object can be changed.

2. If we want to make sure that the value of `vbl` cannot be changed by dereferencing `ptr`, we can write it in two ways:

```
const Type* ptr = &vbl;
const Type* ptr(&vbl);
```

In this case, the addressed object is a constant but the pointer is not.

In addition, if we want to impose both kinds of protection we can write:

```
const Type* const ptr = &vbl;
const Type* const ptr(&vbl);
```

Here is a good way to remember which is which: Read each of the following definitions from right to left (starting with the defined variable).

```
const char * x = &p;           /* x is a pointer to const char*/
char * const y = &q;          /* y is a const pointer to char */
const char * const z = &r;    /* z is a const pointer to a const char */
```

A short program that demonstrates the two kinds of protection is shown in Example 1.22.

EXAMPLE 1.22 `src/constptr/constptr.cpp`

```
#include <iostream>
using namespace std;

int main() {
    int m1(11), m2(13);
    const int* n1(&m1);
    int* const n2(&m2);
    // First snapshot
    cout << "n1 = " << n1 << '\t' << *n1 << '\n'
         << "n2 = " << n2 << '\t' << *n2 << endl;
    n1 = &m2;
    /*n1 = 15; ❶
    m1 = 17; ❷
```

continued

```

//n2 = &m1; ❸
*n2 = 16; ❹
// Second snapshot
cout << "n1 = " << n1 << '\t' << *n1 << '\n'
    << "n2 = " << n2 << '\t' << *n2 << endl;
return 0;
}

```

Output:

```

src/constptr> g++ constptr.cpp
src/constptr> ./a.out
n1 = 0xbffff504 11
n2 = 0xbffff500 13
n1 = 0xbffff500 16
n2 = 0xbffff500 16
src/constptr>

```

- ❶ error: assignment of read-only location
- ❷ m2 is an ordinary int variable—OK to assign
- ❸ error: assignment of read-only variable 'n2'
- ❹ okay to change target

Figure 1.2 shows two snapshots of memory that may help to clarify what is happening when the program runs. Notice that the program produces a memory leak.

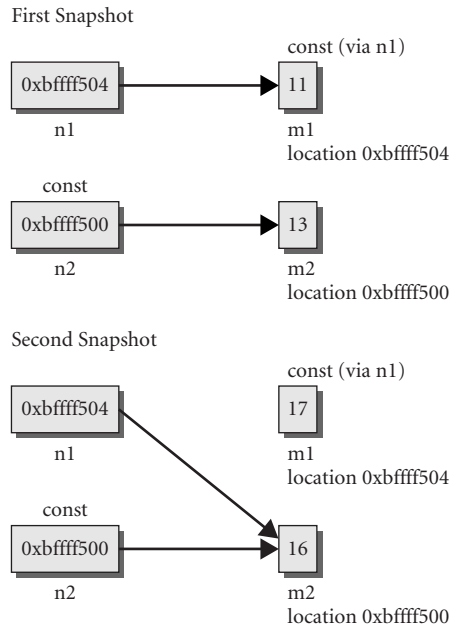


FIGURE 1.2 Two snapshots of memory showing what the program in Example 1.2 produces

An object that is read-only when accessed through one pointer may be changeable when accessed through another pointer. This fact is commonly exploited in the design of functions.

```
char* strcpy(char* dst, const char* src); // strcpy cannot change *src
```

It is okay to assign the address of a variable to a pointer to `const`. It is an error to assign the address of a `const` object to an unrestricted (i.e., non-`const`) pointer variable because that would allow the `const` object's value to be changed.

```
int a = 1;
const int c = 2;
const int* p1 = &c;    // OK
const int* p2 = &a;    // OK
int* p3 = &c;         // error
*p3 = 5;              // error
```

It is good programming practice to use `const` to protect pointer and reference parameters that do not need to be altered by the action of a function. Read-only reference parameters provide the power and efficiency of pass-by-reference with the safety of pass-by-value (see Section 5.6).

1.14 Reference Variables

We observed earlier that an object (in the most general sense) is a contiguous region of storage. An **lvalue** is an expression that refers to an object. Examples of lvalues are variables, array cells, and dereferenced pointers. In essence, an lvalue is anything with a memory address that can be given an alternate name. By contrast, temporary or constant expressions such as `i+1` or `3` are not lvalues.

In C++, a **reference** provides a mechanism for assigning an alternative name to an lvalue. References are useful for avoiding copies when a copy is costly or unnecessary, for example, when passing a very large object as a parameter to a function. A reference must be initialized when it is declared, and the initializer must be an lvalue.

To create a reference to an object of type `SomeType`, a variable must be declared to be of type `SomeType&`. For example:

```
int n;
int& rn = n;
```

The ampersand & following the `int` indicates that `rn` is an **int reference**. The reference variable `rn` is an alias for the actual variable `n`. Note that the & is being used here as a **type modifier** in a declaration, rather than as an operator on an lvalue.

For its entire life, a reference variable will be an alias for the actual lvalue that initialized it. This association cannot be revoked or transferred. For example:

```
int a = 10, b = 20;
int& ra = a;           // ra is an alias for a
ra = b;               // this causes a to be assigned the value 20

const int c = 45;     // c is a constant: its value is read-only.
const int & rc = c;   // legal but probably not very useful.
rc = 10;              // compiler error - const data may not
                    // be changed.
```

The reader has surely noticed that the use of the ampersand in this section might be confused with its use in the earlier section on pointers. To avoid confusion, just remember these two facts:

1. The address-of operator applies to an object and returns its address. Hence, it will only appear on the right side of an assignment or in an initializing expression for a pointer variable.
2. In connection with references, the ampersand is used only in the declaration of a reference. Hence, it will only appear to the left of the reference name as it is being declared.

POINTS OF DEPARTURE

1. See Section 22.1 to learn more about how pointers can be used or misused.
2. See Chapter 19 to learn more about how pointers and types can be converted.

REVIEW QUESTIONS

1. What is a stream? What kinds of streams are there?
2. Give one reason to use an `ostream`.
3. What is the main difference between `getline` and the `>>` operator?
4. What is the type of each expression?
 - a. `3.14`
 - b. `'D'`
 - c. `"d"`
 - d. `6`
 - e. `6.2f`
 - f. `"something stringey"`
 - g. `false`
5. In Example 1.23, identify the type and value of each "thing":

EXAMPLE 1.23 src/types/types.cpp

```
#include <iostream>

int main() {
    using namespace std;
    int i = 5;
    int j=6;
    int *p = &i;           ❶
    int& r=i;
    int& rpr=(*p);
    i = 10;
    p = &j;                 ❷
    rpr = 7;               ❸

    r = 8;                 ❹
    cout << "i=" << i << " j=" << j << endl;  ❺
}

```

- ❶ *p: _____
- ❷ *p: _____
- ❸ *p: _____
- ❹ rpr: _____
- ❺ i: _____ j: _____

6. What is the difference between a pointer and a reference?
7. Why does `main(int argc, char* argv[])` sometimes have parameters? What are they used for?

2

CHAPTER 2

Classes

This chapter provides an introduction to classes and objects, and how member functions operate on objects. UML is introduced. `static` and `const` members are explained. Constructors, destructors, copy operations, and friends are discussed.

2.1	Structs	48
2.2	Class Definitions	49
2.3	Member Access Specifiers	51
2.4	Encapsulation	54
2.5	Introduction to UML	54
2.6	Friends of a Class	55
2.7	Constructors	56
2.8	Subobjects	58
2.9	Destructors	60
2.10	The Keyword <code>static</code>	61
2.11	Copy Constructors and Assignment Operators	64
2.12	Conversions	67
2.13	<code>const</code> Member Functions	68

2.1 Structs

In the C language, there is the `struct` keyword, for defining a structured chunk of memory.

EXAMPLE 2.1 `src/structdemo/demostruct.h`

```
[ . . . . ]
struct Fraction {
    int numer, denom;
    string description;
};
[ . . . . ]
```

A structured piece of memory is comprised of smaller chunks of memory of various types and sizes, each chunk accessible by name. Example 2.1 shows the definition of a `struct`.

Example 2.2 shows how we can use a structured chunk of memory (containing subobjects) as a single entity. Each data member (`numer`, `denom`, `description`) is accessible by name.

EXAMPLE 2.2 `src/structdemo/demostruct.cpp`

```
[ . . . . ]
void printFraction (Fraction f) {
    cout << f.numer << "/" << f.denom << endl;
    cout << "    =? " << f.description << endl;
}

int main() {
    Fraction f1;

    f1.numer = 4;
    f1.denom = 5;
    f1.description = "four fifths";
}
```

```
Fraction f2 = {2, 3, "two thirds"}; ❷

f1.numer = f1.numer + 2;
printFraction (f1);
printFraction (f2);
return 0;
}
```

Output:

```
6/5
=? four fifths
2/3
=? two thirds
```

- ❶ Passing a struct by value could be expensive if it has large components.
- ❷ member initialization

2.2 Class Definitions

C++ has another datatype called **class** that is very similar to `struct`. A simple class definition looks like this:

```
class className {
    members
};
```

The first line of the class definition is called the **classHead**. The features of a class include data members, member functions, and access specifiers. Member functions are used to manipulate or manage the data members.

After a class has been defined, the class name can be used as a **type** for variables, parameters, and returns from functions. Variables of a class type are called **objects**, or **instances**, of a class.

Member functions for class `T` specify the behavior of all objects of type `T` and have access to all members of the class. Non-member functions normally manipulate objects indirectly by calling member functions. The set of values of the data members of an object is called the **state of the object**.

To define a class (or any other type) we generally place the definition in a **header file**, preferably with the same name as the class and with the **.h** extension. Example 2.3 shows a class definition defined in a header file.

EXAMPLE 2.3 `src/classes/fraction.h`

```
#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <string>
using namespace std;

class Fraction {
public:
    void set(int numerator, int denominator);
    double toDouble() const;
    string toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

Header files are `#included` in other files by the preprocessor. To prevent a header file from accidentally being `#included` more than once in any compiled file, we **wrap** it with `#ifndef-#define . . . #endif` preprocessor macros (see Section C.1).

Generally, we place the definitions of member functions outside the class definition in a separate **implementation file** with the `.cpp` extension.

The definition of any class member outside the class definition requires a **scope resolution operator** of the form `ClassName::` before its name. The scope resolution operator tells the compiler that the scope of the class extends beyond the class definition and includes the code between the `::` and the closing brace of the function definition.

Example 2.4 is an implementation file that corresponds to Example 2.3.

EXAMPLE 2.4 `src/classes/fraction.cpp`

```
#include "fraction.h"
#include <sstream>

void Fraction::set(int nn, int nd) {
    m_Numerator = nn;
    m_Denominator = nd;
}
```

```

double Fraction::toDouble() const {
    return 1.0 * m_Numerator / m_Denominator;
}

string Fraction::toString() const {
    ostringstream sstr; ❶
    sstr << m_Numerator << "/" << m_Denominator;
    return sstr.str(); ❷
}

```

- ❶ a stream we write to, but that does not get output anywhere
- ❷ convert the stream just written to a string

Every identifier has a **scope** (see Section 20.2), a region of code where a name is “known” and accessible. Class member names have **class scope**. Class scope includes the entire class definition, regardless of where the identifier was declared, as well as inside each member function definition, starting at the `:` and ending at the closing brace of the definition.

So, for example, the members, `Fraction::m_Numerator` and `Fraction::m_Denominator` are visible inside the definitions of `Fraction::toString()` and `Fraction::toDouble()` even though those function definitions are in a separate file.

2.3 Member Access Specifiers

Thus far we have worked with **class definition code** and **class implementation code**. There is a third category of code as it relates to a given class. **Client code** is code that is outside the scope of the class but that *uses* objects of the class. Generally, client code `#includes` the header file that contains the class definition.

We now revisit the `Fraction` class, focusing on its **member access specifiers**. See Example 2.5.

EXAMPLE 2.5 `src/classes/fraction.h`

```

#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <string>
using namespace std;

```

continued

```
class Fraction {
public:
    void set(int numerator, int denominator);
    double toDouble() const;
    string toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

The member access specifiers, `public`, `protected`, and `private`, are used in a class definition to specify where in a program the affected members can be accessed. The following list provides an informal first approximation of the definitions of these three terms. Refinements are contained in footnotes.

- A `public` member can be accessed (using an object of the class¹) anywhere in a program that `#includes` the class definition file.
- A `protected` member can be accessed inside the definition of a member function of its own class, or a member function of a derived class.²
- A `private` member is only accessible by member functions of its own class.³

Accessibility versus Visibility

There is a subtle difference between **accessibility** and **visibility**. In order for a named item to be accessible, it must first be visible (in our scope). Not all visible items are accessible. Accessibility depends on member access specifiers: `public/private/protected`.

Example 2.6 shows client code that demonstrates visibility errors in a variety of ways. This example also focuses on **block scope**, which extends from an opening brace to a closing brace. A variable declared inside a block is visible and accessible only between its declaration and the closing brace. In the case of a function,

¹ `public static` members can be accessed without an object. We discuss this in Section 2.10.

² We discuss derived classes in Chapter 6.

³ Private members are also accessible by `friends` of the class, which we discuss in Section 2.6.

the block that contains the function definition also includes the function's parameter list.

EXAMPLE 2.6 `src/classes/fraction-client.cpp`

```
#include "fraction.h"
#include <iostream>

int main() {
    const int DASHES = 30;
    using namespace std;

    {
        int i;
        for(i = 0; i < DASHES; ++i)
            cout << "=";
        cout << endl;
    }

    // cout << "i = " << i << endl;
    Fraction f1, f2;
    f1.set(3, 4);
    f2.set(11,12);
    // f2.m_Numerator = 12;
    cout << "The first fraction is: " << f1.toString() << endl;
    cout << "\nThe second fraction, expressed as a double is: "
        << f2.toDouble() << endl;
    return 0;
}
```

- ❶ nested scope—inner block
- ❷ error—`i` no longer exists, so it is not visible in this scope.
- ❸ set through a member function
- ❹ error—`m_Numerator` is visible but not accessible.

Now we can describe the difference between `struct` and `class` in C++. Stroustrup defines a `struct` to be a class in which members are by default public, so that

```
struct T { ...
```

means precisely

```
class T {public: ...
```

2.4 Encapsulation

Encapsulation is the first conceptual step in object-oriented programming. It involves

- Packaging data with the functions that can operate on that data in well-named classes
- Providing clearly named and well-documented `public` functions that allow users of the class to do whatever needs to be done with objects of this class
- Hiding implementation details

The set of `public` function prototypes in a class is called its **public interface**.

The set of `non-public` members, as well as the function definitions themselves, comprise the **implementation**.

One immediate advantage of encapsulation is that it permits the programmer to use a consistent naming scheme for the members of classes. For example, there are many different classes for which it might make sense to have a data member that contains the ID of the particular instance. We could adopt the convention of calling such a data member `m_ID` in every class that needs one. Because class member names are not visible outside the class scope, there is no danger of ambiguity if a member name is also used somewhere else in the program.

2.5 Introduction to UML

UML, the Unified Modeling Language, is a language for object-oriented design. We use UML diagrams, because “a picture is worth 1K words.” UML class diagrams can show the important elements of and relationships between classes in a concise and intuitive way. UML is much more than just class diagrams. Other kinds of UML diagrams can illustrate how classes collaborate with one another and how users interact with classes. We will only use a small subset of UML in this book.

Most of our diagrams were created with a design tool called Umbrello.⁴ For a good overview of UML, we recommend *The Umbrello UML Modeller Handbook*, available from the help menu of Umbrello. Another reference that provides maximum content and minimal bulk is [Fowler04].

⁴ <http://uml.sourceforge.net>

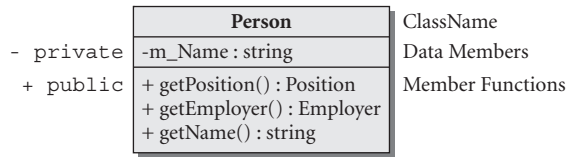


FIGURE 2.1 Person class

Figure 2.1 is a class diagram with only one class: `Person`. Notice that the declarations appear in the *name : type*, Pascal-style, rather than the more familiar C++/Java style, where the names come after the types. This is to help with readability—because we tend to read from left to right, this syntax helps us find names faster. Notice also that `public` members are preceded by a plus sign (+) and `private` members are preceded by a minus sign (-).

2.5.1 UML Relationships

UML is especially good at describing relationships between classes. Looking ahead to Example 2.9, we describe the subobject relationship in UML. Because the subobjects are strictly part of the parent object and cannot possibly exist as stand-alone objects, we could use the **composition** relationship. As Figure 2.2 shows, the filled-in diamond indicates that the object on that side is *composed* (partially) of the object(s) from the other side of the relationship.

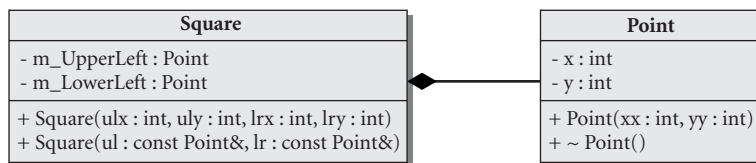


FIGURE 2.2 Composition

2.6 Friends of a Class

Now that we know about accessibility rules, we need to know how to break them occasionally. The `friend` mechanism makes it possible for a class to allow non-member functions to access its private data. The keyword `friend` is followed by

a class or a function declaration. `friend` declarations are located inside a class definition. Here are some syntax examples.

```
class Timer {
    friend class Clock;
    friend void Time::toString();
    friend ostream& operator <<(ostream& os, const Timer& obj);
    [ . . . ]

private:
    long m_Elapsed;
};
```

A `friend` can be a class, a member function of another class, or a non-member function. In the previous example, class `Clock` is a `friend`, so all of its member functions can access `Timer::m_Elapsed`. `Time::toString()` is a `friend` of `Timer` and is assumed (by the compiler) to be a valid member of class `Time`. The third `friend` is a non-member function, an overloaded **insertion operator**, which inserts its second argument into the output stream and returns a reference to the stream so that the operation can be chained.

Breaking encapsulation can compromise the maintainability of your programs, so you should use the `friend` mechanism sparingly and carefully. Typically, `friend` functions are used for two purposes.

1. For factory methods when we want to enforce creational rules (see Section 16.1.4) on a class
2. For global operator functions such as `<<` and `>>` when we do not wish to make the operator a member function, or do not have write-access to the class definition

2.7 Constructors

A **constructor**, sometimes abbreviated as `ctor`, is a special member function that controls the process of object initialization. Each constructor must have the same name as its class. Constructors do not return anything and do not have return types.

There is a special syntax for constructor definitions:

```
ClassName::ClassName( parameter_list )
    :init-list ❶
    {
        constructor body
    }
```

❶ optional

Between the closing parenthesis of the parameter list and the opening brace of a function body, an optional member initialization list can be given. A **member initialization list** begins with a colon (:) and is followed by a comma-separated list of member initializers, each of the form

```
member_name(initializing_expression)
```

If (and only if) no constructor is specified in a class definition, the compiler will supply one that looks like this:

```
ClassName::ClassName()
{ }
```

A constructor that can be called with no arguments has the name **default constructor**. We say that a default constructor gives default initialization to an object of its class. Any data member that is not explicitly initialized in the member initialization list of a constructor is given default initialization.

Classes can have several constructors, each of which initializes in a different (and presumably useful) way. Example 2.7 has three constructors.

EXAMPLE 2.7 `src/ctor/complex.h`

```
class Complex {
public:
    Complex(double realPart, double imPart);
    Complex(double realPart);
    Complex();
private:
    double m_R, m_I;
};
```

Example 2.8 shows the implementation with some client code.

EXAMPLE 2.8 `src/ctor/complex.cpp`

```
#include "complex.h"
#include <iostream>
using namespace std;

Complex::Complex(double realPart, double imPart)
    : m_R(realPart), m_I(imPart) ❶
{
    cout << "complex(" << m_R << ", " << m_I << ")" << endl;
}
```

continued

```

Complex::Complex(double realPart) {
    Complex(realPart, 0);           ❷
}

Complex::Complex() : m_R(0.0), m_I(0.0) {

}

int main() {
    Complex C1;
    Complex C2(3.14);
    Complex C3(6.2, 10.23);
}

```

- ❶ member initialization list
 - ❷ Call one constructor from another, java-style.
-

The default constructor for this class gives default initialization to the two data members of the object `C1`. That initialization is the same kind that would be given to a pair of variables of type `double` in the following code fragment:

```

double x, y;
cout << x << '\t' << y << endl;

```

What would you expect to be the output of that code?

2.8 Subobjects

An object can contain another object, in which case the contained object is considered to be a **subobject**. In Example 2.9, each `Square` object has two `Point` subobjects. Notice that `Point` has a function named `~Point`. We discuss this kind of function in the next section.

EXAMPLE 2.9 `src/subobject/subobject.h`

```

[ . . . . ]
class Point {
public:
    Point(int xx, int yy) : m_x(xx), m_y(yy){}
    ~Point() {
        cout << "point destroyed: ("
            << m_x << ", " << m_y << ")" << endl;
    }
private:
    int m_x, m_y;
};

```

```

class Square {
public:
    Square(int ulx, int uly, int lrx, int lry)
        : m_UpperLeft(ulx, uly), m_LowerRight (lrx, lry)
    {}

    Square(const Point& ul, const Point& lr) :
m_UpperLeft(ul), m_LowerRight(lr) {}
private:
    Point m_UpperLeft, m_LowerRight;
};

[ . . . . ]

```

- ❶ Initialization is required because there is no default ctor.
- ❷ Initialize using the implicitly generated Point copy ctor.
- ❸ embedded subobjects

Whenever an instance of `Square` is created, each of its subobjects is created with it, so that the three objects occupy contiguous chunks of memory.

The `Square` is *composed* of two `Point` objects.

Because `Point` has no default constructor, we must properly initialize for each `Point` subobject in the member initialization list of `Square` (see Example 2.10).

EXAMPLE 2.10 `src/subobject/subobject.cpp`

```

#include "subobject.h"

int main() {
    Square sq1(3,4,5,6);
    Point p1(2,3), p2(8, 9);
    Square sq2(p1, p2);
}

```

Even though no destructor was defined for `Square`, each of its `Point` subobjects is properly destroyed whenever the containing object is. This is an example of **composition**.

```

point destroyed: (8,9)
point destroyed: (2,3)
point destroyed: (8,9)
point destroyed: (2,3)
point destroyed: (5,6)
point destroyed: (3,4)

```

2.9 Destructors

A **destructor**, sometimes abbreviated as `dtor`, is a special member function that automates clean-up actions *just before* an object is destroyed.



WHEN IS AN OBJECT DESTROYED?

- When a local (automatic) object goes out of scope (e.g., when a function call returns)
- When an object created by the `new` operator is specifically destroyed by the use of the operator `delete`
- Just before the program terminates, all objects with `static` storage are destroyed

The destructor's name is the classname preceded by the tilde (`~`) character. It has no return type and no parameters, so it cannot be overloaded. If the class definition contains no destructor definition, the compiler will supply one that looks like this:

```
ClassName::~~ClassName()  
{ }
```

We will look at a less trivial example of a destructor in the next section.



WHEN DO WE NEED TO WRITE A DESTRUCTOR? In general, a class that directly manages or shares an external resource (opens a file, opens a network connection, creates a process, etc.) needs to free the resource at some appropriate time. Such classes are usually wrappers that are responsible for object cleanup.

Qt's container classes make it easy for us to avoid writing code that directly manages dynamic arrays.

You do **not** need a destructor if your class:

- Has simple type members that are not pointers
- Has class members with properly defined destructors themselves

The default compiler-generated destructor calls the destructors on each of its class members in the order that they are listed in the class definition before the object is destroyed. It does *nothing* with pointers or simple types.

2.10 The Keyword `static`

The keyword `static` can be applied to a variable declaration to give the variable **static storage class** (see Section 20.3).

A local `static` variable is created only once, the first time its declaration statement is processed by the running program. It is destroyed when the program terminates. A `static` data member of a class is created once, just before the program begins execution, and is destroyed when the program terminates.

A `static` data member is a piece of data that is associated with the class itself rather than one that belongs to a particular object. It does not affect the `sizeof()` of an object of the class. Each object of a class maintains its own set of non-`static` data members, but there is only one instance of any `static` data member, and it can be shared by all objects of the class.

`static` class members are preferable to (and can generally replace the use of) global variables because they do not pollute the global namespace.

`static` data members must be declared `static` in (and only in) the class definition.

A class member function that does not in any way access the non-`static` data members of the class can (and should) be declared `static`. In Example 2.11, the `static` data member is a private counter that keeps track of the number of `Thing` objects that exist at any given moment. The public `static` member function displays the current value of the `static` counter.

EXAMPLE 2.11 `src/statics/static.h`

```
[ . . . . ]
class Thing {
public:
    Thing(int a, int b);
    ~Thing();
    void display() const ;
    static void showCount();
private:
    int m_First, m_Second;
    static int sm_Count;
};
[ . . . . ]
```

The class UML diagram for `Thing` is shown in Figure 2.3. Notice that the `static` members are underlined in the diagram.

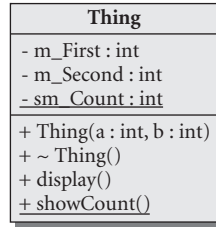


FIGURE 2.3 Thing class diagram

Each `static` data member must be initialized (defined) outside the class definition, preferably in the class implementation file (not a header file) as shown in Example 2.12.⁵

EXAMPLE 2.12 `src/statics/static.cpp`

```
#include "static.h"
#include <iostream>

int Thing::sm_Count = 0; ❶

Thing::Thing(int a, int b)
    : m_First(a), m_Second(b) {
    ++sm_Count;
}

Thing::~Thing() {
    --sm_Count;
}

void Thing::display() const {
    using namespace std;
    cout << m_First << "$$" << m_Second;
}

void Thing::showCount() { ❷
    using namespace std;
    cout << "Count = " << sm_Count << endl;
}
```

- ❶ Must initialize static member!
- ❷ static function

⁵ The exception to this rule is a `static const int`, which can be initialized in the class definition.



Notice that the term `static` does not appear in the **definitions** of `sm_Count` or `showCount()`. This is because the keyword `static` would mean something quite different there: It would change the scope of the variable from global to file-scope (see Section 20.2).

Block-Scope static

`statics` that are defined inside a function or a block of code are initialized when they are executed for the first time.

```
long nextNumber() {
    int localVar(24);
    static long nnumber = 1000;
    cout << nnumber + localVar;
    return ++nnumber;
}
```

The first call to `nextNumber()` initializes `localVar` to 24 and `nnumber` to 1000, displays 1124 on the screen, and returns 1001. When the function returns, `localVar` is destroyed but `nnumber` is not destroyed. Each time that this function is called, `localVar` gets created and initialized to 24 again. The static variable `nnumber` persists between calls and holds onto the value that it obtained in the last call. So, for example, the next time the function is called, 1025 is displayed and 1002 is returned.

static Initialization

A `static` object that is not defined in a block or function is initialized when its corresponding object module (see Section 7.1) is loaded for the first time. Most of the time, this is at program startup, before `main()` starts. The order in which modules get loaded and variables get initialized is implementation dependent.

A `static` object is constructed once and persists until the program terminates. A `static` data member is a `static` object that has class scope.

In Example 2.13, we make use of an internal block so that we can introduce some local objects that will be destroyed before the program ends.

EXAMPLE 2.13 `src/statics/static-test.cpp`

```

#include "static.h"

int main() {
    Thing::showCount(); ❶
    Thing t1(3,4), t2(5,6);
    t1.showCount();     ❷
    {                   ❸
        Thing t3(7,8), t4(9,10);
        Thing::showCount();
    }                   ❹

    Thing::showCount();
    return 0;
}

```

- ❶ No objects exist at this point.
- ❷ direct access through object
- ❸ An inner block of code is entered.
- ❹ end inner block

Here is the compile and run.

```

src/statics> g++ -ansi -pedantic -Wall static.cpp static-test.cpp
src/statics> ./a.out
Count = 0
Count = 2
Count = 4
Count = 2
src/statics>

```

2.11 Copy Constructors and Assignment Operators

C++ gives almost god-like powers to the designer of a class. Object “life cycle” management means taking complete control over the behavior of objects during birth, reproduction, and death. We have already seen how constructors manage the birth of an object and how destructors are used to manage the death of an object. This section investigates the reproduction process: the use of copy constructors and assignment operators.

A **copy constructor** is a constructor that has a prototype like this:

```

ClassName(const ClassName & x);

```

The purpose of a copy constructor is to create an object that is an exact copy of an existing object of the same class.

An **assignment operator** for a class overloads the symbol = and gives it a meaning that is specific to the class. There is one particular version of the assignment operator that has the following prototype:

```
ClassName& operator=(const ClassName& x);
```

Because it is possible to have several different overloaded versions of the operator=() in a class, we call this particular version the **copy assignment operator** (see Example 2.14).

EXAMPLE 2.14 `src/lifecycle/copyassign/fraction.h`

```
[ . . . . ]
class Fraction {
public:
    Fraction(int n, int d) : m_Numer(n), m_Denom(d) {
        ++ctors;
    }
    Fraction(const Fraction& other)
        : m_Numer(other.m_Numer), m_Denom(other.m_Denom) {
        ++copies;
    }
    Fraction& operator=(const Fraction& other) {
        m_Numer = other.m_Numer;
        m_Denom = other.m_Denom;
        ++assigns;
        return *this;
    }

    Fraction multiply(Fraction f2) {
        return Fraction (m_Numer*f2.m_Numer, m_Denom*f2.m_Denom);
    }

    static void report();
private:
    int m_Numer, m_Denom;
    static int assigns;
    static int copies;
    static int ctors;
};
[ . . . . ]
```

The version of `Fraction` in Example 2.14 defines three `static` counters, so that we can count the total number of times each member function is called. This should help us better understand when objects are copied. Example 2.15 uses it to create, copy, and assign some objects.

EXAMPLE 2.15 `src/lifecycle/copyassign/copyassign.cpp`

```
#include <iostream>
#include "fraction.h"

int main() {
    using namespace std;
    Fraction twothirds(2,3);           ❶
    Fraction threequarters(3,4);      ❷
    Fraction acopy(twothirds);        ❸
    Fraction f4 = threequarters;
    cout << "after declarations" ;
    Fraction::report();
    f4 = twothirds;                   ❹
    cout << "before    multiply" ;
    Fraction::report();
    f4 = twothirds.multiply(threequarters);  ❺
    cout << "after    multiply" ;
    Fraction::report();
    return 0;
}
```

- ❶ using 2-arg constructor
- ❷ using copy constructor
- ❸ also using copy constructor
- ❹ assignment
- ❺ Lots of objects get created here.

```
src/ctor> g++ -ansi -pedantic -Wall copyassign.cpp
src/ctor> ./a.out
after declarations
[ assigns: 0 copies: 2 ctors: 2 ]
before    multiply
[ assigns: 1 copies: 2 ctors: 2 ]
after    multiply
[ assigns: 2 copies: 3 ctors: 3 ]
src/ctor>
```



As you can see, the call to `multiply` creates three `Fraction` objects. Can you explain why?

It is important to know that the compiler will supply default versions of the copy constructor and/or the copy assignment operator if one or both are missing from the class definition. The compiler-supplied default versions have the following prototypes for a class `T`:

```
T::T(const T& other);
T& T::operator=(const T& other);
```

2.12 Conversions

A constructor that can be called with a single argument (of a different type) is a **conversion constructor** because it defines a **conversion** from the argument type to the constructor's class. See Example 2.16.

EXAMPLE 2.16 `src/ctor/conversion/fraction.cpp`

```
class Fraction {
public:
    Fraction(int n, int d = 1)           ❶
        : m_Numerator(n), m_Denominator(d) {}
private:
    int m_Numerator, m_Denominator;
};
int main() {
    Fraction frac(8);                   ❷
    Fraction frac2 = 5;                 ❸
    frac = 9;                           ❹
    frac = (Fraction) 7;                ❺
    frac = Fraction(6);                 ❻
    frac = static_cast<Fraction>(6);    ❼
    return 0;
}
```

- ❶ default argument
- ❷ conversion constructor call
- ❸ copy init (calls conversion ctor too)
- ❹ conversion followed by assignment
- ❺ C-style typecast (deprecated)
- ❻ explicit temporary, also a C++ typecast
- ❼ preferred ANSI style typecast

In Example 2.16, the `Fraction` variable `frac` is initialized with a single `int`. The matching constructor is the two argument version, but the second parameter (denominator) defaults to 1. Effectively, this converts the integer 8 to the fraction 8/1.

The *conversion constructor* for `ClassA` is automatically called when an object of that class is required and when such an object can be created by that constructor from the value of `TypeB` that was supplied as an initializer or assigned value.

For example, if `frac` is of type `Fraction` as defined above, then we can write the statement

```
frac = 19;
```

Since `19` is not a `Fraction` object, the compiler checks to see whether it can be converted to a `Fraction` object. Since we have a conversion constructor, this is indeed possible.

So under the covers, the statement `f = 19` makes two “implicit” calls to `Fraction` member functions:

1. `Fraction::Fraction(19);` to convert from `int` to `Fraction`.
2. `Fraction::operator=()` to perform the assignment.

This causes a temporary `Fraction` object on the stack to be created, which is then popped off when the statement is finished executing.

Since we have not defined `Fraction::operator=()`, the compiler uses a default assignment operator that it supplied.

```
Fraction& operator=(const Fraction& fobj);
```

This operator performs a memberwise assignment, from each data member of `fobj` to the corresponding member of the host object.

Ordinarily, any constructor that can be called with a single argument of different type is a conversion constructor that has the implicit mechanisms discussed here. If the implicit mechanisms are not appropriate for some reason, it is possible to suppress them. The keyword **explicit** prevents implicit mechanisms from using that constructor.

2.13 const Member Functions

When a class member function `X::f()` is invoked through an object `x`

```
x.f();
```

we refer to `x` as the **host object**.

The `const` keyword has a special meaning when it is applied to a (non-static) class member function. Placed after the parameter list, `const` becomes part of the function signature and guarantees that the function will not change the state of the host object.

A good way to think of this is to realize that each non-static member function has an implicit parameter, named **this**, which is a pointer to the host object. When we declare a member function to be `const`, we are telling the compiler that, as far as the function is concerned, `this` is a pointer to `const`.

To explain how `const` changes the way a function is invoked, we look at how the original C++ to C preprocessor dealt with member functions.

Since C did not support overloaded functions or member functions, the preprocessor translated the function into a C function with a “mangled” name, which distinguished itself from other functions by encoding the full signature in the name. The mangling process also added an extra implicit parameter to the parameter list: `this`, a pointer to the host object. Example 2.17 shows how member functions might be seen by a linker after a translation into C.

EXAMPLE 2.17 `src/const/constmembers.cpp`

```
#include <iostream>

class Point {
public:
    Point(int px, int py)
        : m_X(px), m_Y(py) {}

    void set(int nx, int ny) { ❶
        m_X = nx;
        m_Y = ny;
    }
    void print() const { ❷
        using namespace std;
        cout << "[" << m_X << ", " << m_Y << "];"
        m_X = 5; ❸
    }
private:
    int m_X, m_Y;
};

int main() {
    Point p(1,1);
    const Point q(2,2);
    p.set(4,4); ❹
    p.print();
    q.set(4,4); ❺
    return 0;
}
```

- ❶ mangled version: `_Point_set_int_int(Point* this, int nx, int ny)`
- ❷ mangled version: `_Point_print_void_const(const Point* this)`
- ❸ Error: `this->m_X = 5`, *this is const.
- ❹ Okay to reassign `p`
- ❺ Error! const object cannot call non-const methods.

In a real compiler, the mangled names for `set` and `print` would be compressed significantly to save space and hence would be less understandable to a human reader.

We can think of the `const` in the signature of `print()` as a modifier of the invisible `this` parameter that points to the host object. This means that the memory pointed to by `this` cannot be changed by the action of the `print()` function. It is a compiler error to change anything via `this` in a `const` member function. The reason that the assignment

```
x = 5;
```

produces an error is that it is equivalent to

```
this->x = 5;
```

The assignment violates the rules of `const`.

EXERCISES: CLASSES

Examples 2.18–2.20 are part of a single program. Use them together to answer the following questions.

1. EXAMPLE 2.18 `src/early-examples/thing.h`

```
#ifndef THING_H_
#define THING_H_

class Thing {
public:
    void set(int num, char c);
    void increment();
    void show();
private:
    int m_Number;
    char m_Character;
};
#endif
```

EXAMPLE 2.19 `src/early-examples/thing.cpp`

```
#include <iostream>
#include "thing.h"

void Thing::set(int num, char c) {
    m_Number = num;
    m_Character = c;
}
```

```
void Thing::increment() {
    ++m_Number;
    ++m_Character;
}

void Thing::show() {
    using namespace std;
    cout << m_Number << '\t' << m_Character << endl;
}
```

EXAMPLE 2.20 `src/early-examples/thing-demo.cpp`

```
#include <iostream>
#include "thing.h"

void display(Thing t, int n) {
    int i;
    for (i = 0; i < n; ++i)
        t.show();
}

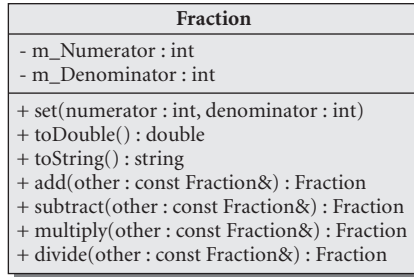
int main() {
    using namespace std;
    Thing t1, t2;
    t1.set(23, 'H');
    t2.set(1234, 'w');
    t1.increment();
    // cout << t1.m_Number;
    display(t1, 3);
    // cout << i << endl;
    t2.show();
    return 0;
}
```

- a.** Uncomment the two commented out lines of code in `thing-demo.cpp` (Example 2.20) and try to compile the program using

```
g++ -ansi -pedantic -Wall thing.cpp thing-demo.cpp
```

Explain the difference between the errors that are reported by the compiler.
 - b.** Add public member functions to the definition of the class `Thing` so that the data members can be kept private and the client code can still output their values.
- 2.** Implement the member functions of this slightly enhanced `Fraction` class from Section 2.2. Given the UML diagram below, define the class, and each

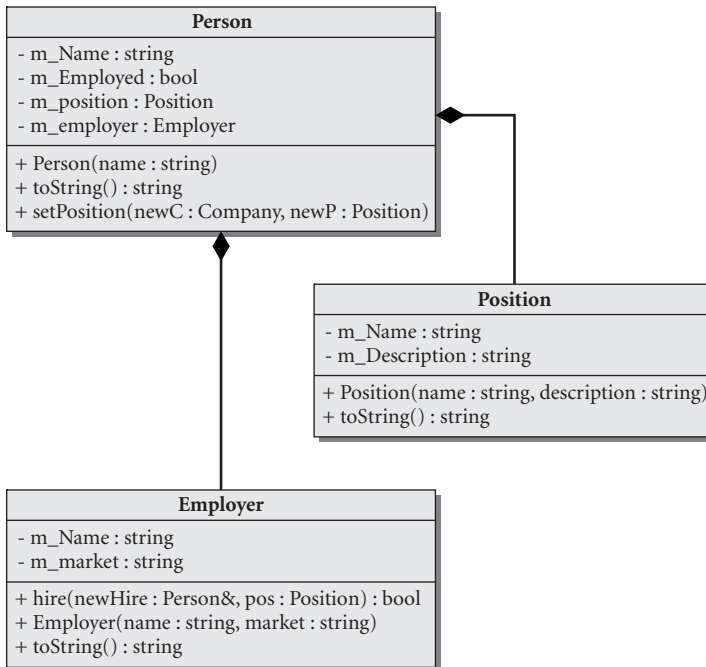
member specified, for an enhanced `Fraction` class. You can use Example 2.4 as a starting point.



Write some client code to test all of the new operations and verify that proper calculations are done.

- Suppose that you wanted to write an application for a company that matches employers and job seekers. A first step would be to design appropriate classes.

The UML diagram, The Company Chart, shows three classes, with `Person` having two subobjects: `Employer` and `Position`.



In order to do this exercise, you will need to use forward declarations (see Section C.1).

- a. Write classes for `Person`, `Position`, and `Employer` as described in The Company Chart.
 - b. For `Person::getPosition()` and `getEmployer()`, create and return something funny if the person has not yet been hired by a company.
 - c. For the `hire(...)` function, set the `Person`'s state so that future calls to `getPosition()` and `getEmployer()` give the correct result.
 - d. In the `main()` program, create at least two `Employers`, the "StarFleet Federation" and the "Borg."
 - e. Create at least two employees, Jean-Luc Picard and Wesley Crusher.
 - f. For each class, write a `toString()` function that gives you a `string` representation of the object.
 - g. Write a main program that creates some objects and then prints out each company's list of employees.
4. Define a class to represent a modern automobile.

Hondurota
- m_Fuel : double - m_Speed : double - m_Odometer : double - sm_FuelTankCapacity : double - sm_FuelConsumptionRate : double
+ Hondurota(odom : double, mpg : double) + getFuel() : double + getSpeed() : double + getOdometer() : double + drive(speed : double, minutes : int) : double + addFuel(gal : double) : double

Here are some features of this class.

- The constructor requires an odometer reading and a value for the average fuel consumption rate in miles per gallon to initialize those two data members.
- The `drive()` function should be reasonably smart:
 - It should not permit the car to drive if there is no fuel.
 - It should adjust the odometer and the fuel amount correctly.
 - It should return the amount of fuel left in the tank.
- The `addFuel()` function should adjust the fuel amount correctly and return the resulting amount of fuel in the tank. `addFuel(0)` should fill the tank.

Write client code to test this class.

5. In the previous problem we really did not make use of the `speed` member. The `drive()` function assumed an average speed and used an average fuel consumption rate. Now let's use the `speed` member to make things a bit more realistic.

Add a member function to the `Hondurota` class that has the prototype

```
double highwayDrive(double distance, double speedLimit);
```

The return value is the elapsed time for the trip.

When driving on a highway, it is usually possible to travel at or near the speed limit. Unfortunately, various things happen that can cause traffic to move more slowly—sometimes much more slowly.

Another interesting factor is the effect that changing speed has on the fuel consumption rate. Most modern automobiles have a speed that is optimal for fuel efficiency (e.g., 45 mph).

Calculating how long it will take to travel a particular distance on the highway, and how much fuel the trip will consume, is the job of this new function.

- Write the function so that it updates the speed, the odometer, and the fuel amount every minute until the given distance has been travelled.
- Use 45 mph as the speed at which fuel is consumed precisely at the stored `sm_FuelConsumptionRate`.
- Use an adjusted consumption rate for other speeds, increasing the rate of consumption by 1% for each mile per hour that the speed differs from 45 mph.
- Your car should stop if it runs out of fuel.
- Assume that you are travelling at the speed limit, except for random differences that you compute each minute by generating a random speed adjustment between -5 mph and $+5$ mph. Don't allow your car to drive faster than 40 mph above the speed limit. Of course, your car should not drive slower than 0 mph. If a random speed adjustment produces an unacceptable speed, generate another one.

Write client code to test this function.

6. Be the computer and predict the output of Example 2.21.

EXAMPLE 2.21 `src/statics/static3.h`

```
#ifndef _STATIC_3_H_
#define _STATIC_3_H_

#include <string>
using namespace std;
```

```

class Client {
public:
    Client(string name) : m_Name(name), m_ID(sm_SavedID++)
        { }
    static int getSavedID() {
        if(sm_SavedID > m_ID) return sm_SavedID;
        else return 0;
    }
    string getName() {return m_Name;}
    int getID() {return m_ID; }
private:
    string m_Name;
    int m_ID;
    static int sm_SavedID ;
};

#endif

```

EXAMPLE 2.22 src/statics/static3.cpp

```

#include "static3.h"
#include <iostream>

int Client::sm_SavedID(1000);

int main() {
    Client cust1("George");
    cout << cust1.getID() << endl;
    cout << Client::getName() << endl;
}

```

7.

Date
- m_DaysSinceBaseDate : unsigned long
+ Date()
+ Date(m : unsigned, d : unsigned, y : unsigned)
+ toString(brief : bool) : string
+ setToToday()
+ getWeekDay() : string
+ lessThan(d : const Date&) : bool
+ equals(d : const Date&) : bool
+ daysAfter(d : const Date&) : int
+ addDays(days : int) : Date
+ leapyear(year : unsigned) : bool
+ monthname(month : unsigned) : string
+ yeardays(year : unsigned) : unsigned
+ monthdays(month : unsigned, year : unsigned) : unsigned

Design and implement a `Date` class subject to the following restrictions and suggestions.

- Each `Date` must be stored as a single integer equal to the number of days since the fixed base date, January 1, 1000 (or some other date if you prefer). Let's call that data member `m_DaysSinceBaseDate`.

- The base year should be stored as a `static unsigned int` (e.g., 1000).
- The class has a constructor and a `set` function that have month, day, year parameters. These three values must be used to compute the number of days from the base date to the given date. We have specified a private member function named `mdy2dsbd()` to do that calculation for both.
- The `toString()` function returns a representation of the stored date in some standard string format that is suitable for display (e.g., `yyyy/mm/dd`). This involves reversing the computation used in the `mdy2dsbd()` function described above. We have specified a private member function named `getMDY()` to do that calculation. We have also suggested a parameter for the `toString()` function (`bool brief`) to allow a choice of date formats.
- We have specified several `static` utility functions (e.g., `leapyear()`) that are `static` because they do not affect the state of this class.
- Make sure you use the correct rule for determining whether or not a given year is a leap year!
- Create a file named `date.h` to store your class definition.
- Create a file named `date.cpp` that contains the definitions of all the functions declared in `date.h`.
- Write client code to test your `Date` class thoroughly.
- Here is the code for `setToToday()` that makes use of the system clock to determine today's date. You will need to `#include <time.h>` (from the C Standard Library) to use this code.

```
void Date::setToToday() {
    struct tm *tp = new(tm);
    time_t now;
    now = time(0);
    tp = localtime(&now);
    set(1 + tp->tm_mon, tp->tm_mday, 1900 + tp->tm_year);
}
```

- `getWeekDay()` function returns the name of the week day corresponding to the stored date. Use this in the fancy version of `toString()`. Hint: January 1, 1900, was a Monday.

8. Consider the class shown in Example 2.23.

EXAMPLE 2.23 `src/destructor/demo/thing.h`

```
#ifndef THING_H_
#define THING_H_

#include <iostream>
#include <string>
using namespace std;
```



```
class Thing {
public:
    Thing(int n) : m_Num(n) {

    }
    ~Thing() {
        cout << "destructor called: "
             << m_Num << endl;
    }

private:
    string m_String;
    int m_Num;
};
#endif
```

The client code in Example 2.24 constructs several objects in various ways and destroys most of them.

EXAMPLE 2.24 `src/destructor/demo/destructor-demo.cpp`

```
#include "thing.h"

void function(Thing t) {
    Thing lt(106);
    Thing* tp1 = new Thing(107);
    Thing* tp2 = new Thing(108);
    delete tp1;
}

int main() {
    Thing t1(101), t2(102);
    Thing* tp1 = new Thing(103);
    function(t1);
    {
        Thing t3(104);
        Thing* tp = new Thing(105);
    }
    delete tp1;
    return 0;
}
```

❶ nested block/scope

Here is the output of this program.

```
destructor called: 107  
destructor called: 106  
destructor called: 101  
destructor called: 104  
destructor called: 103  
destructor called: 102  
destructor called: 101
```

- a.** How many objects were created but not destroyed?
- b.** Why does 101 appear twice in the list?

REVIEW QUESTIONS

1. What is the main advantage of using a `struct`?
2. Describe at least one difference between a `class` and a `struct`.
3. How does class scope differ from block scope?
4. Describe two situations where it is okay to use `friend` functions.
5. How does a `static` data member differ from a `non-static` data member?
6. What is the difference between a `static` member function and a `non-static` member function?
7. What does it mean to declare a member function to be `const`?
8. Explain what would happen (and why) if a class `T` had a copy constructor with the following prototype.

```
T::T(T other);
```

9. Critique the design shown in The Company Chart in Exercise 3. What problems do you see with it? In particular, how would you write `Employer::getEmployees()` or `Position::getEmployer()`?

3

CHAPTER 3

Introduction to Qt

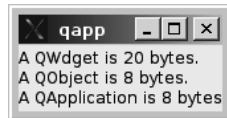
This chapter introduces the Qt development environment, including the compiler, linker, make, and qmake. It includes a first example using Qt, and introduces iterators and lists.

Qt is a modular system of classes and tools that makes it easier for you to write smaller modules of code yourself. This chapter explains how to start reusing Qt.

3.1 Example Project: Using QApplication and QLabel	82
3.2 Makefile, qmake, and Project Files	83
3.3 Getting Help Online	89
3.4 Style Guidelines and Naming Conventions	90
3.5 The Qt Core Module	91
3.6 Streams and Dates	91

3.1 Example Project: Using QApplication and QLabel

Example 3.1 shows a simple main program that creates a `QApplication` and a `QLabel`, manipulates some `QString`s, and then pops up a graphical window.



This example makes use of Qt's `QString` class, which is a dynamic string implementation that supports the Unicode standard.¹ `QString` contains many helpful member functions for converting and formatting strings in different ways. This example also makes use of `QTextStream`, a very flexible Qt class that can provide a well-developed stream interface for reading or writing text to various objects (e.g., text files, strings, and byte arrays).

EXAMPLE 3.1 `src/qapp/main.cpp`

```
#include <QApplication>
#include <QString>
#include <QLabel>
#include <QWidget>
#include <QDebug>
#include <QTextStream>

int main(int argc, char * argv[]) {
    QApplication myapp(argc, argv); ❶
    QWidget wid; ❷
    qDebug() << "sizeof widget: " << sizeof(wid)
             << " sizeof application: " << sizeof(myapp) ;
    QString message;
    QTextStream buf(&message); ❸
    buf << "A QWidget is " << sizeof(wid)
        << " bytes. \nA QObject is " << sizeof(QObject)
```

¹ <http://www.unicode.org/standard/standard.html>

```

        << " bytes. \nA QApplication is " << sizeof(myapp)
        << " bytes.";
    qDebug() << message;
    QLabel label(message);
    label.show();
    return myapp.exec();
};

```

- ❶ All Qt GUI applications need to create one of these at the start of `main()`.
- ❷ We are only creating this to see how big it is.
- ❸ This is a stream that allows us to “write” to the string, similar in usage to `std::stringstream`.
- ❹ Create a GUI widget with the message.
- ❺ Make the label visible.
- ❻ Enter the event loop, and wait for the user to do something. When the user exits, so does `myapp.exec()`.

To build this app, we need a **project file**. A project file describes the project by listing all of the other files, as well as all of the options and file locations that are needed to build the project. Because this is a very simple application, the project file is also simple, as shown in Example 3.2.

EXAMPLE 3.2 `src/qapp/qapp.pro`

```

TEMPLATE = app
SOURCES += main.cpp

```

The first line, `TEMPLATE = app`, indicates that `qmake` should start with a templated `Makefile` suited for building applications. If this project file were for a library, you would see `TEMPLATE = lib` to indicate that a `Makefile` library template should be used instead. A third possibility is that we might have our source code distributed among several subdirectories, each having its own project file. In such a case we might see `TEMPLATE = subdirs` in the project file located in the parent directory, which would cause a `Makefile` to be produced in the parent directory and also in each subdirectory.

3.2 Makefile, qmake, and Project Files

C++ applications are generally composed of many source files, header files, and external libraries. During the normal course of project development, source files and libraries get added, changed, or removed. During the testing/development phase, the project is recompiled and re-linked many times. To produce an executable, all changed source files must be recompiled, and the object files must all be re-linked.

Keeping track of all of the parts of such a project requires a mechanism that precisely specifies the input files involved, the tools needed to build, the intermediate targets and their dependencies, and the final executable target.

The most widely used utility for handling the job of building a project is **make**. It reads the details of the project specifications and the instructions for the compiler from a **Makefile**, which resembles a shell script but contains (at a minimum):

- **Rules** for building certain kinds of files (e.g., to get a `.o` file from a `.cpp` file, we must run `gcc -c` on the `.cpp` file)
- **Targets** that specify which executables (or libraries) must be built
- **Dependencies** that list which targets need to be rebuilt when certain files get changed

The **make** command by default loads the file named `Makefile` from your current working directory and performs the specified build steps (**compiling and linking**).

The immediate benefit of using **make** is that it recompiles only the files that need to be rebuilt, rather than blindly recompiling every source file every time. Figure 3.1 is a diagram that attempts to show the steps involved in building a Qt application. To learn more about **make**, we recommend the book *The Linux Development Platform* by Rafeeq Ur Rehman and Christopher Paul (Prentice Hall).

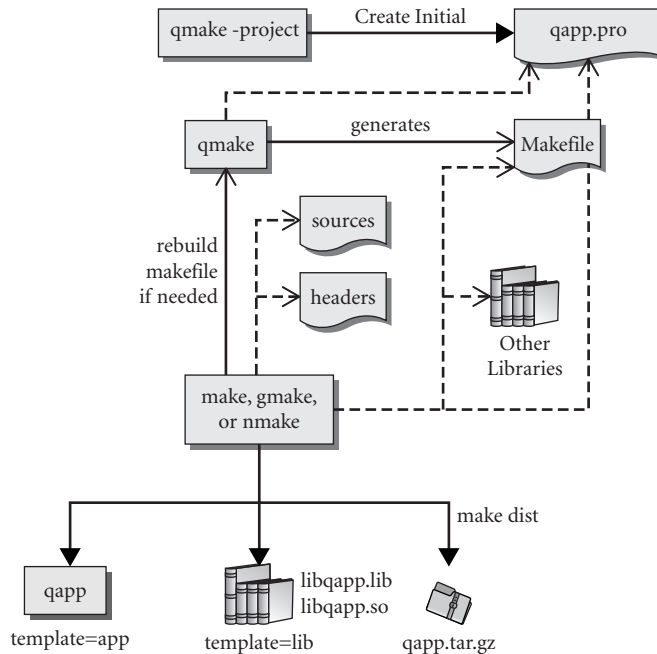


FIGURE 3.1 (q)make build steps

With Qt, it is not necessary to write `Makefiles`. Qt provides a tool called `qmake` to generate `Makefiles` for you. It is still necessary to somehow run `make` and understand its output. Most IDEs run `make` (or something similar) under the covers and either display or filter its output.

The following transcript shows which files get created at each step of the build process for Example 3.1.

```
src/qapp> ls -sF
total 296
  4 main.cpp
src/qapp> qmake -project
src/qapp> ls -sF
total 296
  4 main.cpp  4  qapp.pro
src/qapp> qmake
src/qapp> ls -sF
total 296
  4 main.cpp  4  qapp.pro  4  Makefile
src/qapp> make
g++ -c -pipe -g -Wall -W                # compile step
    -D_REENTRANT -DQT_CORE_LIB -DQT_GUI_LIB -DQT_SHARED
    -I/usr/local/qt/mkspecs/linux-g++ -I.
    -I/usr/local/qt/include/QtGui -I/usr/local/qt/include/QtCore
    -I/usr/local/qt/include -I. -I. -I.
    -o main.o main.cpp
g++ -Wl,-rpath,/usr/local/qt/lib        # link step
    -L/usr/local/qt/lib -L/usr/local/qt/lib -lQtGui_debug
    -L/usr/X11R6/lib -lpng -lXi -lXrender -lXinerama -lfreetype
    -lfontconfig -lXext -lX11 -lm -lQtCore_debug -lz -ldl -lpthread
    -o qapp main.o
src/qapp> ls -sF
total 420
  4 main.cpp      132 main.o      124 qapp*
  8 Makefile     4  qapp.pro
src/qapp>
```

Notice that we can see the arguments passed to the compiler when we run `make`. If any errors are encountered, we will see them too.

3.2.1 #include: Finding Header Files

There are three commonly used ways to `#include` a library header file:

```
#include <headerFile>
#include "headerFile"
#include "path/to/headerFile"
```

The angle brackets (`< >`) indicate that the preprocessor must look (sequentially) in the directories listed in the **include path** for the file.

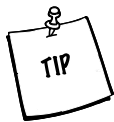
A quoted filename indicates that the preprocessor should look for `headerfile` in the including file's directory first. A quoted path indicates that the preprocessor should check the path directory first. The path information can be absolute or relative (to the including file's directory). If no file is found at the specified location, the directories listed in the include path are searched for `headerfile`.

If versions of the file exist in more than one directory in the include path, the search will stop as soon as the first occurrence of the file has been found. If the file is not found in any of the directories of the search path, then the compiler will report an error.

For items in the C++ Standard Library, the compiler generally already knows where to find the header files. For other libraries, we can expand the search path by adding a `-I/path/to/headerfile` switch to the compiler.

If you use an IDE, there will be a `Project->Settings->Preprocessor`, or `Project->Options->Libraries` configuration menu that lets you specify additional include directories, which end up getting passed as `-I` switches to the compiler.

With `qmake`, as we will soon see, you can add `INCLUDEPATH += dirname` lines to the project file. These directories end up in the generated Makefile as `INCPATH` macros, which *then* get passed on to the compiler/preprocessor at build time.



In general, it is a good idea to `#include` non-Qt header files after Qt header files. Since Qt does define many symbols, in the compiler as well as the preprocessor, this helps you avoid/find name clashes more easily.

3.2.2 The make Command

Instead of running the command line compiler directly, we will start using `make`,² which greatly simplifies the build process when a project involves multiple source files and libraries.

We have seen before that `qmake` (without arguments) reads a project file and builds a Makefile. Example 3.3 is a slightly abbreviated look at the Makefile for the previous `qapp` project.

² Depending on your development environment, this program goes under many other names, such as `mingw32-make`, `nmake`, `gmake`, or `unsermake`.

EXAMPLE 3.3 src/qapp/Makefile-abbreviated

```

# Excerpts from a makefile

##### Compiler, tools and options

CC          = gcc      # executable for C compiler
CXX         = g++     # executable for c++ compiler
LINK        = g++     # executable for linker

# flags that get passed to the compiler
CFLAGS      = -pipe -g -Wall -W -D_REENTRANT $(DEFINES)
CXXFLAGS    = -pipe -g -Wall -W -D_REENTRANT $(DEFINES)
INCPATH     = -I/usr/local/qt/mkspecs/default -I. \
             -I$(QTDIR)/include/QtGui -I$(QTDIR)/include/QtCore \
             -I$(QTDIR)/include

# Linker flags
LIBS        = $(SUBLIBS) -L$(QTDIR)/lib -lQtCore_debug
             -lQtGui_debug -lpthread
LFLAGS      = -Wl,-rpath,$(QTDIR)/lib

# macros for performing other operations as part of build steps:
QMAKE       = /usr/local/qt/bin/qmake

##### Files

HEADERS     =      # If we had some, they'd be here.
SOURCES     = main.cpp
OBJECTS     = main.o
[snip]
QMAKE_TARGET = qapp
DESTDIR     =
TARGET      = qapp # default target to build

first: all          # to build "first" we must build "all"

##### Implicit rules

.SUFFIXES: .c .o .cpp .cc .cxx .C

.cpp.o:
$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<

## Possible targets to build

all: Makefile $(TARGET) # this is how to build "all"

$(TARGET): $(OBJECTS) $(OBJMOC) # this is how to build qapp
$(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(OBJMOC) $(OBJCOMP) \
$(LIBS)

```

continued

```

        qmake: FORCE                # "qmake" is a target too!
        @$(QMAKE) -o Makefile qapp.pro # what does it do?

dist:                                     # Another target
    @mkdir -p .tmp/qapp \
        && $(COPY_FILE) --parents $(SOURCES) $(HEADERS) \
        $(FORMS) $(DIST) .tmp/qapp/ \
        && (cd 'dirname .tmp/qapp' \ && $(TAR) qapp.tar qapp \
            && $(COMPRESS) qapp.tar) \
        && $(MOVE) 'dirname .tmp/qapp'/qapp.tar.gz . \
        && $(DEL_FILE) -r .tmp/qapp

clean:compiler_clean                    # yet another target
    -$(DEL_FILE) $(OBJECTS)
    -$(DEL_FILE) *~ core *.core

##### Dependencies for implicit rules

main.o: main.cpp

```

The command `make` checks the dependencies and performs each build step specified in the Makefile. The name and location of the final result can be set with the project variables, `TARGET` and `target.path`. If `TARGET` is not specified, the name defaults to the name of the directory in which the project file is located. If `target.path` is not specified, the location defaults to the directory in which the project file is located.

3.2.3 Cleaning Up Files

`make` can clean up the generated files for you with the two targets `clean` and `distclean`. Observe how they are different from the following code:

```

src/qapp> make clean
rm -f main.o
rm -f *~ core *.core
src/qapp> ls
Makefile  main.cpp  qapp  qapp.pro

src/qapp> make distclean
rm -f qmake_image_collection.cpp
rm -f main.o
rm -f *~ core *.core
rm -f qapp
rm -f Makefile
src/qapp> ls
main.cpp  qapp.pro

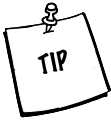
```

After a `make distclean`, the only files that remain are the source files that can go into a tarball for distribution.



If you modify a project file since the last execution of `make`, the next invocation of `make` should rebuild the `Makefile` itself (via `qmake`) before re-running `make` on the newly generated `Makefile`.

In other words, the `Makefile` is `qmake`-aware and can re-`qmake` itself.



The command `make dist` will create a tarball (`dirname.tar.gz`) that contains all the source files that the project file knows about.

As we add more source-code, header, or library modules for our project, we edit the `.pro` file and add the new items to the `SOURCES`, `HEADERS`, and `LIBS` lists. The same documentation standards that apply to C++ source code should be applied to project files (where comments begin with `#`).

We think of the project file as a map of our project, containing references to all files and locations required for building our application or library. Like other source code files, this is both human readable and machine readable. *The .pro file is the first place to look when we encounter “not found” or “undefined” messages during the build process (especially at link time).* For further details we recommend that you read Trolltech’s guide to `qmake`.³

3.3 Getting Help Online

In addition to the Trolltech Online Documentation,⁴ which includes links to API docs as well as FAQs,⁵ there are many online resources available to you.

1. The Qt Interest Mailing List⁶ provides a developer community and searchable archives. Searching on a class you are trying to use, or an error message you do not understand, will often give you useful results.
2. QtCentre,⁷ a Web-based online community.
3. If it’s not directly related to Qt, and the error message is short and rare enough, Google might even come up with interesting hits.
4. There is also a `#qt` irc channel on `irc.freenode.net`.

³ <http://trolltech.com/qmake-manual.html>

⁴ <http://doc.trolltech.com>

⁵ <http://www.trolltech.com/developer/faqs/>

⁶ <http://lists.trolltech.com/qt-interest/>

⁷ <http://www.qtcentre.org/>

3.4 Style Guidelines and Naming Conventions

C++ is a powerful language that supports many different programming styles. The coding style used in most Qt programs is not “pure” C++. Instead, it uses combination of macros and preprocessor trickery to achieve a higher-level dynamic language that more closely resembles Java or Python than C++. In fact, to take full advantage of Qt’s power and simplicity, we tend to abandon the Standard Library entirely.

We find there are certain aspects to “Qt programming style” that are worth emulating, and they are summarized here. For a more complete guide, see “Designing Qt-Style C++ APIs,” by Matthias Ettrich, published by Trolltech.

- Class names begin with a capital letter: `class Customer`
- Function names begin with a lowercase letter.
- Although permitted by the compiler, periods, underscores, dashes, and funny characters should be *avoided* whenever possible (except where noted below).
- Multi-word names have subsequent words capitalized: `class FileTagger void getStudentInfo()` for example.
- Constants should be in CAPS.
- Each class name should be a noun or a noun phrase: `class LargeFurryMammal;` for example.
- Each function name should be a verb or a verb phrase: `processBookOrder();` for example.
- Each `bool` variable name should produce a reasonable approximation of a sentence when used in an `if()` statement: `bool isQualified;` for example.

For data members, we use a common prefix.

- Member name: `m_Color`, `m_Width` (prepend lowercase `m_`)
- static data members: `sm_Singleton`, `sm_ObjCount`

For each attribute, we have naming conventions for their corresponding getters/setters.

- Non-boolean getters: `color()` or `getColor()`⁸
- Boolean getters: `isChecked()` or `isValid()`
- Setter: `setColor(const Color& newColor);`

⁸ The latter is Java style, the former is Qt style. Both conventions are widely used. Try to be consistent in your code.

A consistent naming convention greatly improves the readability and maintainability of a program.

3.5 The Qt Core Module

Qt 4 is a library consisting of smaller libraries, or **modules**. The most popular ones are

- **core**—including `QObject`, `QThread`, `QFile`, and so forth
- **gui**—all classes derived from `QWidget`, and some related classes
- **xml**—for parsing and serializing XML
- **sql**—for communicating with SQL databases
- **net**—for communicating data between hosts on specific protocols (`http`, `tcp`, `udp`)

Except for **core**, modules need to be “enabled” in `qmake` project files in order to be used. For example:

```
QT += xml # to use the xml module
QT += gui # to use QWidget
QT += sql # to use SQL module
```

The following section will introduce some of the core library classes.

3.6 Streams and Dates

In subsequent examples, we use instances of `QTextStream`, which behave in a similar way to the C++ Standard Library’s global `iostream` objects. We have given them the familiar names: `cin`, `cout`, and `cerr`. For convenience, we have placed these definitions, along with some other useful functions, into a namespace so that they can be easily added to any program.

EXAMPLE 3.4 `src/libs/utils/qstd.h`

```
[ . . . . ]
namespace qstd {
    extern QTextStream cin;      ❶
    extern QTextStream cout;
    extern QTextStream cerr;
    bool yes(QString yesNoQuestion);
    bool more(QString prompt);
    int promptInt(int base = 10);
    double promptDouble();
}
```

continued

```
void promptOutputFile(QFile& outfile);
void promptInputFile(QFile& infile);
```

[. . . .]

❶ declared only—defined in the .cpp file

Example 3.4 declares the `iostream`-like `QTextStreams`, and Example 3.5 contains the required definitions of these `static` objects.

EXAMPLE 3.5 `src/libs/utlils/qstd.cpp`

[. . . .]

```
QTextStream qstd::cin(stdin, QIODevice::ReadOnly);
QTextStream qstd::cout(stdout, QIODevice::WriteOnly);
QTextStream qstd::cerr(stderr, QIODevice::WriteOnly);
```

`<QTextStream>` works with Unicode `QStrings` and other Qt types, so we will use it in favor of `<iostream>` in most of our examples henceforth. The program in Example 3.6 uses `QTextStream` objects and functions from the `qstd` namespace just described. It also uses some of the `QDate` member functions and displays dates in several different formats.

EXAMPLE 3.6 `src/qtio/qtio-demo.cpp`

[. . . .]

```
int main() {
    using namespace qstd;
    QDate d1(2002, 4,1), d2(QDate::currentDate());
    int days;
    cout << "The first date is: " << d1.toString()
         << "\nToday's date is: "
         << d2.toString("ddd MMMM d, yyyy") << endl;

    if (d1 < d2)
        cout << d1.toString("MM/dd/yy") << " is earlier than "
             << d2.toString("yyyyMMdd") << endl;

    cout << "There are " << d1.daysTo(d2)
         << " days between "
         << d1.toString("MMM dd, yyyy") << " and "
         << d2.toString(Qt::ISODate) << endl;

    cout << "Enter number of days to add to the first date: "
         << flush;
    days = promptInt();
```



```

cout << "The first date was " << d1.toString()
      << "\nThe computed date is "
      << d1.addDays(days).toString() << endl;
cout << "First date displayed in longer format: "
      << d1.toString("dddd, MMMM dd, yyyy") << endl;
[ . . . . ]

```

Here is the output of this program.

```

The first date is: Mon Apr 1 2002
Today's date is: Wed January 4, 2006
04/01/02 is earlier than 20060104
There are 1374 days between Apr 01, 2002 and 2006-01-04
Enter number of days to add to the first date: : 1234
The first date was Mon Apr 1 2002
The computed date is Wed Aug 17 2005
First date displayed in longer format: Monday, April 01, 2002

```

EXERCISE: THE QT CORE MODULE

- Write a birthday reminder application called `birthdays`. Classes to reuse are `QDate`, `QFile`, `QString`, `QStringList`, and `QTextStream`.
 - Store name/birthday pairs in any format you like, in a file called `birthdays.dat`.
 - `birthdays` with no command line arguments opens the file and lists all birthdays coming up in the next 30 days, in chronological order.
 - `birthdays -a "john smith" "yyyy-mm-dd"` should add an entry to the file.
 - `birthdays -n 40` shows birthdays coming up in the next 40 days.
 - `birthdays namespec` searches for the birthday paired with the name *namespec*.

POINTS OF DEPARTURE

- Visit the Unicode Web site. Explain what it is and why it is important that `QString` supports the Unicode standard.
- Explore the `QString` documentation. Explain what it means for a string implementation to support the Unicode standard.
- Look up `make` in the documentation. Discuss four useful command line options for the `make` command.

REVIEW QUESTIONS

1. What is a project file? How can you produce one for your project?
2. What does the `TEMPLATE` variable mean in the `qmake` project file? What are possible values?
3. What is a Makefile? How can you produce a Makefile for your project?

4

CHAPTER 4

Lists

Whenever possible, we use lists in favor of arrays. This chapter explains ways of grouping things together in lists and how to iterate through them.

4.1 Introduction to Containers	96
4.2 Iterators	97
4.3 Relationships	99

4.1 Introduction to Containers

There are many occasions when it is necessary to deal with collections of things. The classic approach in languages like C is to use arrays to store such collections. In C++ arrays are regarded as evil. Here are a few good reasons to avoid using arrays.

- Array subscripts are not checked to make sure that they are not out of range. A programmer using an array has the responsibility to write extra code to do the range checking.
- Arrays are either fixed in size or they must use dynamic memory from the heap. With heap arrays, the programmer is responsible for making sure that, under all possible circumstances, the memory gets properly deallocated when the array is destroyed. To do this properly requires deep knowledge of C++, exceptions, and what happens under exceptional circumstances.
- Inserting, prepending, or appending elements to an array can be expensive operations (in terms of both run time and developer time).

The Standard Library and Qt both provide the programmer with *lists* that resize themselves as needed and also perform range checking. `std::list` and `QList` are each considered basic **generic containers** in their respective libraries. They are similar to each other in **interface** (the way they are used from client code), but very different in **implementation** (the way they behave at runtime).

A *generic container* is named as such because

1. **Generics** are classes or functions that accept template (see Section 10.1) parameters so that they can operate on any type.
2. **Containers** (see Section 10.2) are objects that can *contain* other objects.

To use a `QList`, the client code must contain a declaration that answers the question: “List of what?” Like other generic containers, `QList` is a **template** class (see Chapter 10) and must be declared in the following way.

```
QList<double>  doublList;  
QList<Thing>  thingList;
```

`QList` supports many operations. As with any class you reuse, it is recommended that you scan the API docs to get an overview of its full capabilities. With a single function call, items can be added, removed, swapped, queried, cleared, moved, located, and counted in a variety of ways.

4.2 Iterators

Any time you have a container of things, sooner or later you are probably going to loop through the container and do something with each thing. An **iterator** is an object that provides indirect access to each element in a container. It is specifically designed to be used in a loop.

Qt 4 supports the following styles of iteration:

1. Qt 4 style `foreach` loops, similar to Perl and Python
2. Java 1.2 style `Iterator`
3. Standard Library style `ContainerType::iterator`
4. Hand-made `while` or `for` loops that use getters of the container

The next section demonstrates the various styles of iteration available in C++ with Qt 4.

4.2.1 `QStringList` and Iteration

For text processing, it is very useful to work with lists of strings. `QStringList` is *derived* from `QList<QString>` so it *inherits* all of `QList`'s behavior (see Chapter 6). In addition, `QStringList` has some string-specific convenience functions such as `indexOf()`, `join()`, and `replaceInStrings()`.

Converting between lists and individual strings is quite easy with perl-like `split()` and `join()` functions. Example 4.1 demonstrates lists, iterations, `split()`, and `join()`.

EXAMPLE 4.1 `src/collections/lists/lists-examples.cpp`

```
#include <QStringList>
#include <QDebug>
#include <cassert>

/* Some simple examples using QStringList, split and join */
```

continued

```

int main() {

    QString winter = "December, January, February";
    QString spring = "March, April, May";
    QString summer = "June, July, August";
    QString fall = "September, October, November";

    QStringList list;
    list << winter;           ❶
    list += spring;          ❷
    list.append(summer);     ❸
    list << fall;

    qDebug() << "The Spring months are: " << list[1] ;

    QString allmonths = list.join(", ");
    /* from list to string - join with a ", " delimiter */
    qDebug() << allmonths;

    QStringList list2 = allmonths.split(", ");
    /* split is the opposite of join. Each month will have its
    own element. */

    assert(list2.size() == 12);           ❹

    foreach (QString str, list) {        ❺
        qDebug() << QString(" [%1] ").arg(str);
    }

    for (QStringList::iterator it = list.begin();
        it != list.end(); ++it) {       ❻
        QString current = *it;           ❼
        qDebug() << "[" << current << "]"";
    }

    QListIterator<QString> itr (list2);   ❽
    while (itr.hasNext()) {              ❾
        QString current = itr.next();
        qDebug() << "{" << current << "}";
    }

    return 0;
}

```

- ❶ append operator 1
- ❷ append operator 2
- ❸ append member function
- ❹ Assertions abort the program if the condition is not satisfied.
- ❺ Qt 4 foreach loop, similar to Perl/Python and Java 1.5 style for loops.
- ❻ C++ STL-style iteration
- ❼ pointer-style dereference
- ❽ Java 1.2 style iterator
- ❾ Java iterators point in between elements.

```

src/collections> ./collections
The Spring months are: March, April, May
December, January, February, March, April, May, June, July, August,
September, October, November
[December]
[January]
[February]
[March]
[April]
[May]
[June]
[July]
[August]
[September]
[October]
[November]
src/collections>

```

4.3 Relationships

When there is a **one-to-one** or a **one-to-many** correspondence between objects of certain types, we can describe a **relationship** between them in a UML class diagram.

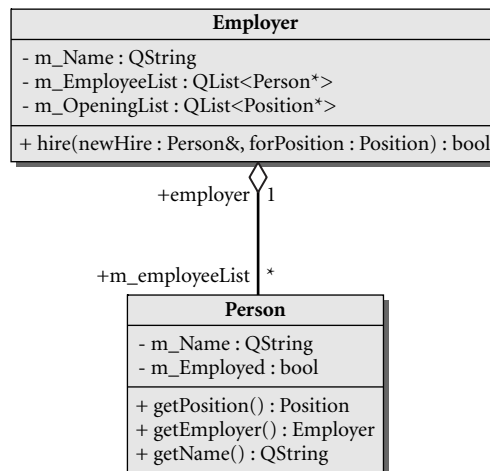


FIGURE 4.1 Simple relationship

In the relationship shown in Figure 4.1, we see a diamond on the `Employer` side that can be read to mean “From the `Employer`, there is an `employeeList` containing `Persons`.”

Since there could be many `Person` instances working for the `Employer`, there is a `*` on the `Person` end of the relationship. This is a one to many relationship between `Employer` and `Person`, where `*` takes on its regular expression definition, of “0 or more” of something (see Chapter 13).

Revisiting the company UML diagram from The Company Chart in Chapter 2, Exercise 3, another set of relationships, as shown in Figure 4.2, is revealed when we look at the company from the `Employer`’s viewpoint. These relationships are

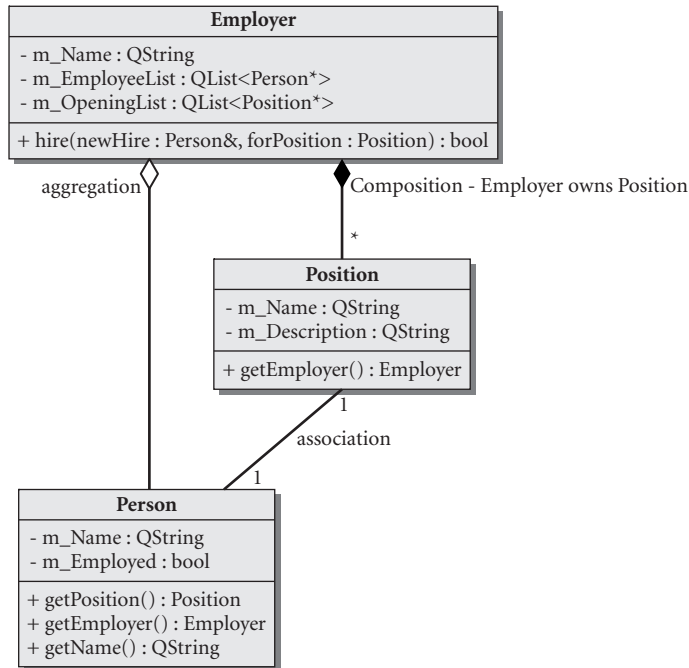


FIGURE 4.2 The employer’s view of the company

1. There is a **composition relationship** between `Employer` and `Position`. This indicates that the `Employer` *owns* the `Position`, and the `Position` should not exist without the `Employer`.
2. There is an **aggregate relationship** (the hollow diamond) from the `Employer` to its employees. The `Employer` groups together a collection of `Persons` and gets them to do things during working hours. In an aggregate relationship, the lifetimes of the objects on either end are not related to each other.

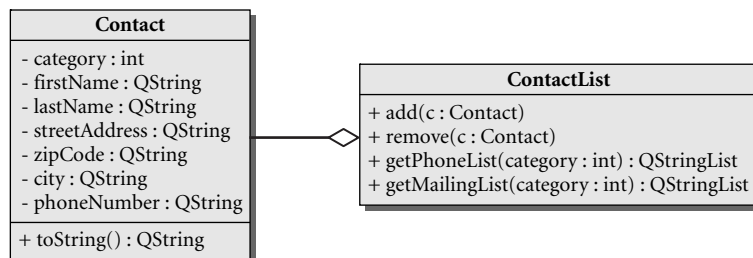
3. The `Person` and the `Position` have an **association** between them. An association is a bidirectional relationship and does not specify anything about ownership or management between objects. Although our diagram indicates a 1:1 relationship between them, it is possible that other `Employers` many hire many `Persons` for the same `Position`. In such a case, the numbers should reflect that in the diagram.

EXERCISES: RELATIONSHIPS

In these exercises you are to implement some relationships based on Figure 4.2. The diagram is only a starting point—you will need to add some members to the classes.

1. Implement the `findJobs()` function to look at an `Employer`, and get a list of all open `Positions`.
2. Implement the `apply()` function to call `Employer::hire()`, and return the same result as `hire`, if successful.
3. Have the `Employer::hire()` function randomly return `false` half of the time, to make things interesting.
4. Create some more test `Employer` objects (Galactic Empire and Klingon Empire), `Person` objects (Darth Vader, C3PO, Data), and `Position` objects (Tie Fighter Pilot, Protocol Android, Captain) in your client code.
5. Make up some funny job application scenarios, and run your program to determine whether they are successful.

EXERCISES: CONTACT LIST



1. This UML diagram describes a data model for a contact system. `ContactList` can derive from or reuse any Qt container that you wish, as long as it supports the operations listed.

- `getPhoneList(int category)` accepts a value to be compared with a `Contact`'s `category` member for selection purposes. It returns a `QStringList` containing, for each selected `Contact`, the name and phone number, separated by the tab symbol: `"\t"`.
 - `getMailingList()` has a similar selection mechanism and returns a `QStringList` containing address label data.
2. Write a `ContactFactory` class that generates random `Contacts`. Example 4.2 contains a substantial hint.

EXAMPLE 4.2 `src/containers/contact/testdriver.cpp`

```
[ . . . . ]

void createRandomContacts(ContactList& cl, int n=10) {
    static ContactFactory cf;
    for (int i=0; i<n; ++i) {
        cf >> cl; ❶
    }
}
```

❶ adds a `Contact` into the `ContactList`

There are many ways to generate random names/addresses. One way is to have the `ContactFactory` create lists of typical first names, last names, street names, city names, and so forth. When it is time to generate a contact, it can pick a random element from each list, add randomly generated address numbers, zip codes, and so on. We demonstrate the use of the random function in "Exercise: C++ Simple Types" in Chapter 1.

3. Write client code to test your classes. In particular, the client code should generate some random contacts, try out the queries, and print the original list, as well as the query results, to the standard output. Summarize the results by listing the number of elements in the original `ContactList` compared to the query results.

POINTS OF DEPARTURE

1. Read the API docs for `QList` and find three distinct ways of adding elements to the list.
2. List three methods that exist in `QStringList` but are not in `QList`.
3. Why does `QList` have an iterator and an `Iterator`? What is the difference between them?

REVIEW QUESTIONS

1. What is an iterator?
2. Draw a UML diagram with three or more classes that have at least one of each kind of relationship: aggregation and composition. The classes should represent real-world concepts, and the relationships should attempt to represent reality. Write a couple of paragraphs explaining why there is a relationship of each kind there.

5

CHAPTER 5

Functions

In this chapter we discuss the essentials of function declarations, prototypes, and signatures; overloading functions; function call resolution; default/optional arguments; temporary variables and when they're created; reference parameters and return values; and inline functions.

5.1	Function Declarations	106
5.2	Overloading Functions	107
5.3	Optional Arguments	109
5.4	Operator Overloading	111
5.5	Parameter Passing by Value	116
5.6	Parameter Passing by Reference	118
5.7	References to const	121
5.8	Function Return Values	122
5.9	Returning References from Functions	122
5.10	Overloading on const-ness	124
5.11	Inline Functions	126
5.12	Inlining versus Macro Expansion	127

5.1 Function Declarations

Functions in C++ are very similar to functions and subroutines in other languages. C++ functions, however, support many features that are not found in some languages, so it is worthwhile discussing each of them here. Each function has:

1. A name
2. A return type (which may be void)
3. A parameter list (which may be empty)
4. A body (a group of statements)

The first three are the function's interface, and the last is the implementation.

A function must be *declared* before it is used for the first time. The mechanism for declaring a function is called a **function prototype**. A function prototype is a declaration that must include

- The function's return type
- The function's name
- An ordered, comma separated list of the types of the function's parameters

Here are a few prototypes.

```
int toCelsius(int fahrenheitValue);
QString toString();
int main(int argc, char* argv[]);
```

Even though parameter names are optional in function prototypes, it is good programming practice to use them. They constitute a very effective and efficient part of the documentation for a program. Furthermore, many documentation generators (e.g., kdoc) require them.

A simple example can help to show why parameter names should be used in function prototypes. Suppose we needed a constructor for a `Date` class that we designed and we wanted that constructor to initialize the `Date` with values for the year, month, and day. If we presented the prototype as `Date(int, int, int)`, the user of that class would not know immediately what order to list the three

values when constructing a `Date` object. Since several of the possible orderings are in common use somewhere on the planet, there is no “obvious” answer that would eliminate the need for more information. By simply naming the parameters the problem is eliminated and the function has documented itself.

In multi-file applications, function prototypes are usually stored in header files.

5.2 Overloading Functions

The **signature** of a function consists of its name and its parameter list. In C++, the return type is *not* part of the signature.

C++ permits **overloading** of function names. A function name is overloaded if it has more than one meaning within a given scope. Overloading occurs when two or more functions within a given scope have the same name but different signatures. It is an error to have two functions in the same scope with the same signature but different return types.

Function Call Resolution

When a function call is made to an overloaded function within a given scope, the C++ compiler determines from the arguments which version of the function to invoke. To do this, a match must be found between the number and type of the arguments and the signature of exactly one of the overloaded functions. This is the sequence of steps that the compiler takes to determine which overloaded function to call.

1. If there is an exact match with one function, call it.
2. Else, match through standard type promotions (see Section 19.5).
3. Else, match through conversion constructors or conversion operators (see Section 2.12).
4. Else, match through *ellipsis* (`...`) (see Section 24.1), if found
5. Else, report an error.

Example 5.1 shows a class with six member functions, each with a distinct signature. Keep in mind that each function has an additional implicit parameter: `this`. The keyword `const`, following the parameter list, protects the host object from the action of the function and is part of its signature.

EXAMPLE 5.1 `src/functions/function-call.cpp`

```
[ . . . . ]

class SignatureDemo {
public:
    SignatureDemo(int val) : m_Val(val) {}
    void demo(int n)
        {cout << ++m_Val << "\tdemo(int)" << endl;}
    void demo(int n) const ❶
        {cout << m_Val << "\tdemo(int) const" << endl;}
/* void demo(const int& n)
    {cout << ++m_Val << "\tdemo(int&)" << endl;} */ ❷
    void demo(short s)
        {cout << ++m_Val << "\tdemo(short)" << endl;}
    void demo(float f)
        {cout << ++m_Val << "\tdemo(float)" << endl;}
    void demo(float f) const
        {cout << m_Val << "\tdemo(float) const" << endl;}
    void demo(double d)
        {cout << ++m_Val << "\tdemo(double)" << endl;}
private:
    int m_Val;
};
```

- ❶** overloaded on const-ness
 - ❷** clashes with previous function
-

Example 5.2 contains some client code that tests the overloaded functions from `SignatureDemo`.

EXAMPLE 5.2 `src/functions/function-call.cpp`

```
[ . . . . ]

int main() {
    SignatureDemo sd(5);
    const SignatureDemo csd(17);
    sd.demo(2);
    csd.demo(2); ❶
    int i = 3;
    sd.demo(i);
    short s = 5;
    sd.demo(s);
    csd.demo(s); ❷
    sd.demo(2.3); ❸
    float f(4.5);
    sd.demo(f);
    csd.demo(f);
    csd.demo(4.5);
    return 0;
}
```


- ❶ const version is called.
- ❷ Non-const short cannot be called, so a promotion to int is required to call the const int version.
- ❸ This is double, not float.

The output should look something like this:

```
6      demo(int)
17     demo(int) const
7      demo(int)
8      demo(short)
17     demo(int) const
9      demo(double)
10     demo(float)
17     demo(float) const
```

EXERCISES: OVERLOADING FUNCTIONS

1. Experiment with Example 5.1. Start by uncommenting the third member function and compiling.
2. Try adding the following line just before the end of `main()`:

```
csd.demo(4.5);
```

What happened? Explain the error message.
3. Add other function calls and other variations on the `demo()` function. Explain each result.

5.3 Optional Arguments

Function parameters can have default values, making them optional. The default value for an optional argument can be a constant expression or an expression that does not involve local variables.

Parameters with default arguments must be the right-most (trailing) parameters in the parameter list.

Trailing arguments with default values can be left out of the function call. The corresponding parameters will then be initialized with the default values.

From the viewpoint of the function, if it is called with one missing argument, then that argument must correspond to the last parameter in the list. If two arguments are missing, they must correspond to the last two parameters in the list (and so forth).

Because an optional argument specifier applies to a function's interface, it belongs with the declaration, not the definition of the function if the declaration is kept in a separate header file.

A function with default arguments can be called in more than one way. If all arguments for a function are optional, the function can be called with no arguments. Declaring a function with n optional arguments can be thought of as an abbreviated way of declaring $n+1$ functions, one for each possible way of calling the function.

In Example 5.3 the constructor for the `Date` class has three parameters; each parameter is optional and defaults to 0.

EXAMPLE 5.3 `src/functions/date.h`

```
[ . . . . ]
class Date {
public:
    Date(int d = 0, int m = 0, int y = 0);
    void display(bool eoln = true) const;
private:
    int m_Day, m_Month, m_Year;
};
[ . . . . ]
```

The constructor definition shown in Example 5.4 looks the same as usual; no default arguments need to be specified there.

EXAMPLE 5.4 `src/functions/date.cpp`

```
#include <QDate>
#include "date.h"
#include <iostream>

Date::Date(int d , int m , int y )
: m_Day(d), m_Month(m), m_Year(y) {

    static QDate currentDate = QDate::currentDate(); ❶

    if (m_Day == 0) m_Day = currentDate.day();
    if (m_Month == 0) m_Month = currentDate.month();
    if (m_Year == 0) m_Year = currentDate.year();
}

void Date::display(bool eoln) const {
    using namespace std;
    cout << m_Year << "/" << m_Month << "/" << m_Day;
    if (eoln)
        cout << endl;
}
```

❶ We use Qt's `QDate` class only to get the current date.

If 0 is the actual value of any of the supplied arguments, it will be replaced with a sensible value, derived from the current date.

EXAMPLE 5.5 `src/functions/date-test.cpp`

```
#include "date.h"
#include <iostream>

int main() {
    using namespace std;
    Date d1;
    Date d2(15);
    Date d3(23, 8);
    Date d4(19, 11, 2003);

    d1.display(false);
    cout << '\t';
    d2.display();
    d3.display(false);
    cout << '\t';
    d4.display();
    return 0;
}
```

The client code in Example 5.5 demonstrates that by defining default values we are, in effect, overloading the function. The different versions of the function execute the same code, but with different values passed in for the later parameters.

So if we ran this program on November 26, 2005, it should show us this in the output:

```
src/functions> qmake
src/functions> make
[ compiler linker messages ]
src/functions> ./functions
11/26/2005      11/15/2005
8/23/2005      11/19/2003
src/functions>
```

5.4 Operator Overloading

The keyword `operator` is used in C++ to define a new meaning for an operator symbol such as `+`, `-`, `=`, `*`, `&`, and so forth. Adding a new meaning to an operator symbol is a specialized form of function overloading.

Operator overloading provides a more compact syntax for calling functions, which can lead to more readable code (assuming the operators are used in ways that are commonly understood).

It is possible to overload nearly all of the existing operator symbols in C++. For example, suppose that we want to define a class named `Complex` to represent complex numbers.¹ To specify how to do the basic arithmetic operations with these objects we could overload the four arithmetic operator symbols. While we are at it, we could also overload the insertion symbol so that output statements become more readable.

Example 5.6 shows a class definition with both members and non-member operators.

EXAMPLE 5.6 `src/complex/complex.h`

```
#include <iostream>
using namespace std;

class Complex {
    // binary non-member friend function declarations
    friend ostream& operator<<(ostream& out, const Complex& c);
    friend Complex operator-(const Complex& c1, const Complex & c2);
    friend Complex operator*(const Complex& c1, const Complex & c2);
    friend Complex operator/(const Complex& c1, const Complex & c2);

public:
    Complex(double re = 0.0, double im = 0.0);

    // binary member function operators
    Complex& operator+= (const Complex& c);
    Complex& operator-= (const Complex& c);

    Complex operator+(const Complex & c2); ❶

private:
    double m_Re, m_Im;
};
```

❶ This should be a non-member friend like the other non-mutating operators.

¹ Complex numbers were introduced initially to describe the solutions to equations such as $x^2 - 6x + 25 = 0$. Using the quadratic formula one can easily determine that the roots of this equation are $3 + 4i$ and $3 - 4i$. The complex numbers consist of all numbers of the form $a + bi$, where a and b are real numbers and i is the square root of -1 . Since that set includes such numbers for which $b = 0$, it is clear that the real numbers are a subset of the complex numbers.

The operators declared in Example 5.6 are all binary (accept 2 operands). For the member functions, there is only one formal parameter because the first (left) operand is implicit: `*this`. The member operators definitions are shown in Example 5.7.

EXAMPLE 5.7 `src/complex/complex.cpp`

```
[ . . . . ]

Complex& Complex::operator+=(const Complex& c) {
    m_Re += c.m_Re;
    m_Im += c.m_Im;
    return *this;
}

Complex Complex::operator+(const Complex& c2) {
    return Complex(m_Re + c2.m_Re, m_Im + c2.m_Im);
}

Complex& Complex::operator-=(const Complex& c) {
    m_Re -= c.m_Re;
    m_Im -= c.m_Im;
    return *this;
}
```

Example 5.8 shows the definitions of the non-member friend functions. They are defined like ordinary global functions.

EXAMPLE 5.8 `src/complex/complex.cpp`

```
[ . . . . ]

ostream& operator<<(ostream& out, const Complex& c) {
    out << '(' << c.m_Re << ',' << c.m_Im << ')' ;
    return out;
}

Complex operator-(const Complex& c1, const Complex& c2) {
    return Complex(c1.m_Re - c2.m_Re, c1.m_Im - c2.m_Im);
}
```

We have expressed the mathematical rules that define each of the four algebraic operations in C++ code. These details are encapsulated and hidden so that client code does not need to deal with them. Example 5.9 shows some client code that demonstrates and tests the `Complex` class.

EXAMPLE 5.9 src/complex/complex-test.cpp

```

#include "complex.h"
#include <iostream>

int main() {
    using namespace std;
    Complex c1(3.4, 5.6);
    Complex c2(7.8, 1.2);

    cout << c1 << " + " << c2 << " = " << c1 + c2 << endl;
    cout << c1 << " - " << c2 << " = " << c1 - c2 << endl;
    Complex c3 = c1 * c2;
    cout << c1 << " * " << c2 << " = " << c3 << endl;
    cout << c3 << " / " << c2 << " = " << c3 / c2 << endl;
    cout << c3 << " / " << c1 << " = " << c3 / c1 << endl;

    return 0;
}

```

Here is the output of the program in Example 5.9:

```

(3.4,5.6) + (7.8,1.2) = (11.2,6.8)
(3.4,5.6) - (7.8,1.2) = (-4.4,4.4)
(3.4,5.6) * (7.8,1.2) = (19.8,47.76)
(19.8,47.76) / (7.8,1.2) = (3.4,5.6)
(19.8,47.76) / (3.4,5.6) = (7.8,1.2)

```

Member versus Global Operators

As we have seen, it is possible to overload operators as member functions, or as global functions. The primary difference that you will notice first is how they can be called. In particular, a member function operator *requires* an object as the left operand. A global function, in contrast, permits the same kinds of type conversions for either operand.

Example 5.10 shows why `Complex::operator+()` would be better suited as a non-member function.

EXAMPLE 5.10 src/complex/complex-conversions.cpp

```

#include "complex.h"

int main() {
    Complex c1 (4.5, 1.2);
    Complex c2 (3.6, 1.5);
}

```

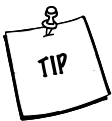
```
Complex c3 = c1 + c2;  
Complex c4 = c3 + 1.4; ❶  
Complex c5 = 8.0 - c4; ❷  
Complex c6 = 1.2 + c4; ❸  
}
```

- ❶ Right operand is promoted.
- ❷ Left operand is promoted.
- ❸ Error: Left operand is not promoted for member operators.

There are some limitations on operator overloading. Only built-in operators can be overloaded. It is not possible to introduce definitions for symbols such as \$, ", or ' that do not already possess operator definitions. Although new meanings can be defined for built-in operators, their associativity and precedence remain the same.

It is possible to overload all of the built-in binary and unary operators except for these:

- The ternary conditional operator `testExpr ? valueIfTrue : valueIfFalse`
- Scope resolution operator `::`
- Member select operators `.` and `.*`



Here is one way to remember which operators can be overloaded: If it has a dot in it (.) anywhere, it's probably not allowed.



Overloading the comma operator is allowed, but not recommended until you are a C++ expert.

We have provided a complete table of operator symbols and their characteristics in Section 19.1.

EXERCISES: OPERATOR OVERLOADING

1. Continue the development of the `Fraction` class by adding overloaded operators for addition, subtraction, multiplication, division, and various kinds of comparison. In each case the parameter should be a `const Fraction&`. Write client code to test your new operators.
2. To be really useful, a `Fraction` object should be able to interact with other kinds of numbers. Expand the definition of `Fraction` so that the operators in Exercise 1 also work for `int` and `double`. It should be clear, for example, how to get `frac + num` to be correctly evaluated. How would you handle the expression `num + frac`, where `frac` is a `Fraction` and `num` is an `int`? Write client code to test your new functions.
3. Add arithmetic and comparison operators to the class `Complex`. Write client code to test your expanded class.

5.5 Parameter Passing by Value

By default, C++ parameters are passed by value. When a function is called, a temporary (local) copy of each argument object is made and placed on the program stack. Only the local copy is manipulated inside the function, and the argument objects from the calling block are not affected by these manipulations. The temporary stack variables are all destroyed when the function returns. A useful way to think of value parameters is this: **Value parameters** are merely local variables that are initialized by copies of the corresponding argument objects that are specified when the function is called. Example 5.11 gives a short demonstration.

EXAMPLE 5.11 `src/functions/summit.cpp`

```
#include <iostream>

int sumit(int num) {
    int sum = 0;
    for (; num ; --num) ❶
        sum += num;
    return sum;
}

int main() {
    using namespace std;
    int n = 10;
    cout << n << endl;
    cout << sumit(n) << endl;
    cout << n << endl; ❷
    return 0;
}
```


Output :

```
10
55
10
```

- ❶ The parameter gets reduced to 0.
 - ❷ See what `sumit()` did to `n`.
-

If a pointer is passed to a function, a temporary copy of that pointer is placed on the stack. Changes to that pointer will have no effect on the pointer in the calling block. For example, the temporary pointer could be assigned a different value (see Example 5.12).

EXAMPLE 5.12 `src/functions/pointerparam.cpp`

```
#include <iostream>
using namespace std;

void messAround(int* ptr) {
    *ptr = 34;           ❶
    ptr = 0;            ❷
}

int main() {
    int n(12);          ❸
    int* pn(&n);        ❹
    cout << "n = " << n << "\tpn = " << pn << endl;
    messAround(pn);    ❺
    cout << "n = " << n << "\tpn = " << pn << endl;
    return 0;
}
```

Output :

```
n = 12   pn = 0xbffff524
n = 34   pn = 0xbffff524
```

- ❶ Change the value that is pointed to.
 - ❷ Change the address stored by `ptr`.
 - ❸ Initialize an int.
 - ❹ Initialize a pointer that points to `n`.
 - ❺ See what is changed by `messAround()`.
-

In the output we display the hexadecimal value of the pointer `pn` as well as the value of `n` so that there can be no doubt about what was changed by the action of the function.

5.6 Parameter Passing by Reference

Large objects, or objects with expensive copy constructors, should not be passed by value because the creation of temporary copies consumes time, machine cycles, and memory needlessly. In C, we passed objects by pointer to avoid copying them. However, using pointers requires a different syntax from using regular variables. Further, accidental misuse of pointers can cause data corruption, leading to runtime errors that can be very difficult to find and fix. In C++ (and C99), we can pass by reference, which offers the same performance as a pointer-pass. With objects, this permits use of the `(.)` operator for accessing members.

A **reference parameter** is simply a parameter that is an alias for something else. To declare a parameter to be a reference, put the ampersand character (`&`) between the type name and the parameter name.

A reference parameter of a function is initialized by the actual argument being passed when the function is called. That argument must be, as with any reference, a non-`const` lvalue. Changes to a non-`const` reference parameter in the function cause changes to the actual object used to initialize the parameter. This feature is often exploited to allow functions, which can return at most one value, to cause changes in several objects, effectively allowing the function to return several values. Example 5.13 shows how reference parameters can be used with integers.

EXAMPLE 5.13 `src/reference/swap.cpp`

```
#include <iostream>
using namespace std;

void swap(int &a, int &b) {
    int temp = a;
    cout << "Inside the swap() function:\n"
         << "address of a: " << &a
         << "\taddress of b: " << &b
         << "\naddress of temp: " << &temp << endl;
    a = b;
    b = temp;
}

int main() {
    int n1 = 25;
    int n2 = 38;
    int n3 = 71;
    int n4 = 82;
    cout << "Initial values:\n"
         << "address of n1: " << &n1
         << "\taddress of n2: " << &n2
         << "\nvalue of n1: " << n1
         << "\t\t\tvalue of n2: " << n2
```

```

    << "\naddress of n3: " << &n3
    << "\taddress of n4: " << &n4
    << "\nvalue of n3: " << n3
    << "\t\t\tvalue of n4: " << n4
    << "\nMaking the first call to swap()" << endl;
swap(n1,n2);
cout << "After the first call to swap():\n"
    << "address of n1: " << &n1
    << "\taddress of n2: " << &n2
    << "\nvalue of n1: " << n1
    << "\t\t\tvalue of n2: " << n2
    << "\nMaking the second call to swap()" << endl;
swap(n3,n4);
cout << "After the second call to swap():\n"
    << "address of n3: " << &n3
    << "\taddress of n4: " << &n4
    << "\nvalue of n3: " << n3
    << "\t\t\tvalue of n4: " << n4 << endl;
return 0;
}

```

There are extra output statements in Example 5.13 to help keep track of the addresses of the important variables.

```

Initial values:
address of n1: 0xbffff3b4      address of n2: 0xbffff3b0
value of n1: 25                value of n2: 38
address of n3: 0xbffff3ac      address of n4: 0xbffff3a8
value of n3: 71                value of n4: 82

```

Initially our stack might look something like Figure 5.1:

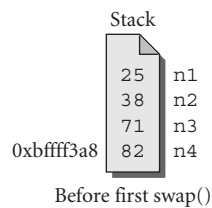


FIGURE 5.1 Before first swap()

As the program proceeds, we will see output like this:

```

Making the first call to swap()
Inside the swap() function:
address of a: 0xbffff3b4      address of b: 0xbffff3b0
address of temp: 0xbffff394

```

When references get passed to functions, the values that get pushed onto the stack are *addresses*, not values. Under the covers, pass-by-reference is very much like pass-by-pointer. Our stack now might look like Figure 5.2:

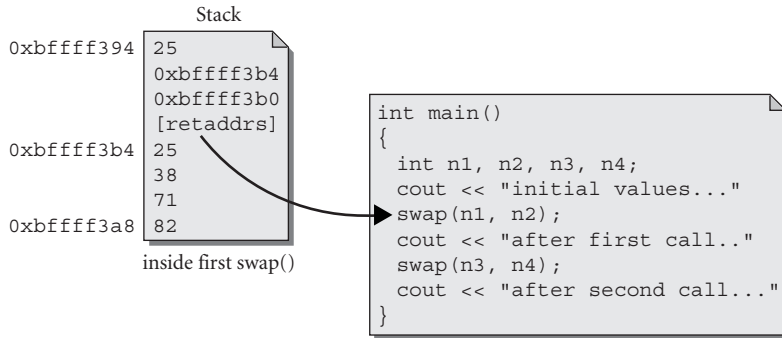


FIGURE 5.2 Inside first swap()

After the first call to `swap()`:

address of `n1`: `0xbffff3b4` address of `n2`: `0xbffff3b0`
 value of `n1`: 38 value of `n2`: 25

Making the second call to `swap()`

Inside the `swap()` function:

Now our stack might look like Figure 5.3:

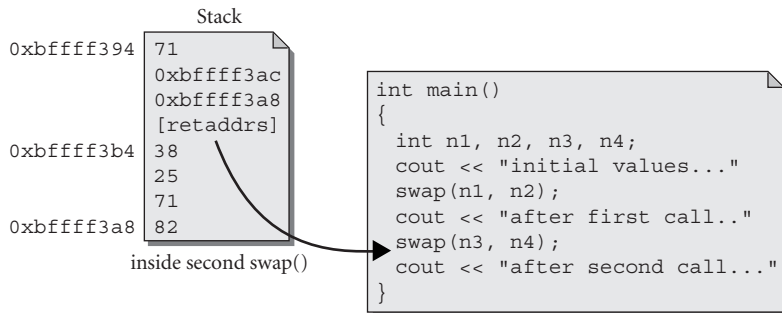


FIGURE 5.3 Inside second swap()

address of `a`: `0xbffff3ac` address of `b`: `0xbffff3a8`

address of `temp`: `0xbffff394`

After the second call to `swap()`:

address of `n3`: `0xbffff3ac` address of `n4`: `0xbffff3a8`
 value of `n3`: 82 value of `n4`: 71

The `swap()` function is actually working with `n1` and `n2` during the first call and with `n3` and `n4` during the second call.

Pass-by-reference is an alternate syntax for passing by pointer. Under the covers, it is implemented the same way (i.e., the value is not copied). The main difference between pass-by-pointer and pass-by-reference is that you must de-reference a pointer, while a reference is accessible in the same way as the referred “thing.”



PASS-BY-POINTER OR PASS-BY-REFERENCE? When you have a choice, it is generally preferable to use references instead of pointers because this can reduce the number of places where a programmer can accidentally corrupt memory. It is only when you need to manage objects (creation, destruction, adding to a managed container) that you need to operate on pointers, and those routines can usually be encapsulated as member functions.

5.7 References to const

Declaring a reference parameter to be `const` tells the compiler to make sure that the function does not attempt to change that object. For objects larger than a pointer, a reference to `const` is an efficient alternative to a value parameter because no data is copied. Example 5.14 contains three functions, each accepting a parameter in a different way.

EXAMPLE 5.14 `src/const/reference/constref.cpp`

```
class Person {
public:
    void setNameV( QString newName) {
        newName += " Smith"; ❶
        m_Name = newName;
    }

    void setNameCR( const QString& newName) {
//      newName += " Python"; ❷
        m_Name = newName;
    }
    void setNameR( QString& newName) {
        m_Name += " Dobbs"; ❸
        m_Name = newName;
    }
private:
    QString m_Name;
};

int main() {
    Person p;
    QString name("Bob");
    p.setNameCR(name); ❹
//  p.setNameR("Monty"); ❺
    p.setNameCR("Monty"); ❻
}
```

continued

```

    p.setNameV("Connie");           7
    p.setNameR(name);               8
    cout << name;
}

```

- ❶ Changes a temporary that's about to be destroyed.
 - ❷ Error: Can't change const&.
 - ❸ Changes the original QString.
 - ❹ No temporaries are created.
 - ❺ Error: Cannot convert to a QString&.
 - ❻ char* converts to temporary and gets passed by const reference.
 - ❼ Temporary QString #1 is created to convert char* to QString. Temporary #2 is created when it is passed by value.
 - ❽ No temporaries are created.
-

5.8 Function Return Values

Some functions return a value when they have finished performing the task for which they were designed. Space for a temporary return object is usually a register (if it can fit), but sometimes it is an object allocated on the stack. The temporary return object is initialized when the `return` statement is executed and exists just long enough to be used in whatever expression contains the function call. It is generally a copy of an object that is local to the function or an object constructed from an expression in the `return` statement.

5.9 Returning References from Functions

Sometimes it can be very useful to design a function so that it returns a reference. For example, when we overload the insertion operator, `operator<<(ostream&, NewType)`, we always return a reference to the output stream. This makes it possible to chain operations like this:

```
cout << thing1 << thing2 << thing3 ... ;
```

A reference return (especially of `*this`) is used to provide lvalue behavior for member functions.

As with reference parameters, it is possible to protect a reference return by specifying that the object it aliases is `const`.

Example 5.15 captures the essence of reference returns.

EXAMPLE 5.15 `src/reference/maxi.cpp`

```

#include <iostream>
using namespace std;

int& maxi(int& x, int& y) {
    return (x > y) ? x : y;
}

int main() {
    int a = 10, b = 20;
    maxi(a,b) = 5;      ❶
    maxi(a,b) += 6;    ❷
    ++maxi(a, b) ;     ❸
    cout << a << '\t' << b << endl;
    return 0;
}

```

Output:

```
17      5
```

- ❶ Assigns the value 5 to b.
- ❷ Increases a by 6. a is now 16.
- ❸ Increments a by 1.

As we see in the `main()` function, the reference return value of the function `maxi()` makes the expression `maxi(a, b)` into a **modifiable lvalue**.



Be very careful that your function does not return a reference to a temporary (local) object. A moment's thought should make that restriction clear: When the function returns, all of its local variables are destroyed.

```

int& max(int i, int j) {
    int retval = i > j ? i : j;

    return retval;
}

```

Code like the above may generate a compiler warning (if you are lucky). Alas, the compiler does not consider it an error.

```
badmax.cpp:4: warning: reference to local variable 'retval' returned
```

A more practical example showing the benefits of reference returns is coming up in Example 5.16, which defines some common operators for vectors.

5.10 Overloading on const-ness

`const` changes the signature of a member function. This means that functions can be overloaded on const-ness. Example 5.16 shows a homemade vector class with member functions overloaded in this way.

EXAMPLE 5.16 `src/const/overload/constoverload.h`

```
#ifndef CONSTOVERLOAD_H
#define CONSTOVERLOAD_H

#include <iostream>

class Point3 {
public:
    friend std::ostream& operator<<(std::ostream& out, const
        Point3& v);
    Point3(double x = 0, double y = 0, double z = 0);
    double& operator[](int index);
    const double& operator[](int index) const;
    Point3 operator+(const Point3& v) const;
    Point3 operator-(const Point3& v) const;
    Point3 operator*(double s) const;
private:
    static const int cm_Dim = 3;
    double m_Coord[cm_Dim];
};

#endif
```

- ❶ a 3D point (of double)
- ❷ overloaded on const-ness
- ❸ scalar multiplication

EXERCISES: OVERLOADING ON CONST-NESS

1. In Example 5.17, the compiler can tell the difference between calls to the `const` and to the non-`const` versions of `operator[]` based on the const-ness of the object.

EXAMPLE 5.17 `src/const/overload/constoverload-client.cpp`

```

#include "constoverload.h"
#include <iostream>

int main( ) {
    using namespace std;
    Point3 pt1(1.2, 3.4, 5.6);
    const Point3 pt2(7.8, 9.1, 6.4);
    double d ;
    d = pt2[2]; ❶
    cout << d << endl;
    d = pt1[0]; ❷
    cout << d << endl;
    d = pt1[3]; ❸
    cout << d << endl;
    pt1[2] = 8.7; ❹
    cout << pt1 << endl;
    // pt2[2] = 'd';
    cout << pt2 << endl;
    return 0;
}

```

❶ _____
 ❷ _____
 ❸ _____
 ❹ _____

- a. Which operator is called for each of the notes?
- b. Why is the last assignment commented out?
- c. The operator function definitions are shown in Example 5.18. The fact that the two function bodies are identical is worth pondering. If `index` is in range, each function returns `m_Coord[index]`, so what is the difference between them?

EXAMPLE 5.18 `src/const/overload/constoverload.cpp`

```

[ . . . . ]
const double& Point3::operator[](int index) const {
    if ((index >= 0) && (index < cm_Dim))
        return m_Coord[index];
    else
        return zero(index);
}

```

continued

```
double& Point3::operator[](int index) {
    if ((index >= 0) && (index < cm_Dim))
        return m_Coord[index];
    else
        return zero(index);
}
[ . . . . ]
```

5.11 Inline Functions

To avoid the overhead associated with a function call (creation of a stack frame containing copies of arguments or addresses of reference parameters and the return address) C++ permits you to declare functions to be **inline**. Such a declaration is a *request* to the compiler that it replace each call to the function with the fully expanded code of the function. For example:

```
inline int max(int a, int b){
    return a > b ? a : b ;
}

int main(){
    int temp = max(3,5);
    etc....
}
```

The compiler could substitute the expanded code for `max` as shown here.

```
int main() {
    int temp;
    {
        int a = 3;
        int b = 5;
        temp = a > b ? a : b;
    }
    etc.....
}
```

The inlining of a function can give a significant boost in performance if it is called repeatedly (e.g., inside a large loop). The penalty for inlining a function is that it might make the compiled code larger, which will cause the program to use more memory while it is running. For small functions that get called many times, that memory effect will be small while the potential performance gain might be large.

There are no simple answers to the question of whether inlining will improve the performance of your program or not. A lot depends on the optimization settings of the compiler. A lot depends on the nature of the program. Does it make very heavy use of the processor? Does it make heavy use of system memory? Does

it spend a lot of its time interacting with slow devices (e.g., input and output)? The answers to these questions affect the answer to the question of whether or not to `inline`, and we leave them to a more advanced treatment of the subject. For an overview of the complexity of this issue, visit Marshall Cline's FAQ Lite site.²

An `inline` function is similar to a `#define` macro with one very important difference: The substitution process for a `#define` macro is handled by the preprocessor, which is essentially a text editor. The substitution process for an `inline` function is handled by the compiler, which will perform the operation much more intelligently with proper type checking. We discuss this in more detail in the next section.

Some Rules about Inline Functions

- An inline function must be defined before it is called (a declaration is not enough).
- An inline definition can only occur once in any source code module.
- If a class member function's definition appears inside the class definition, the function is implicitly inline.

If a function is too complex, or the compiler options are switched, the compiler may ignore the `inline` directive. Most compilers refuse to inline functions that contain:

- `while`, `for`, `do ... while` statements
- `switch` statements
- More than a certain number of lines of code

If the compiler does refuse to inline a function, it treats it as a normal function and generates regular function calls.

5.12 Inlining versus Macro Expansion

Macro expansion is a mechanism for placing code inline by means of the following preprocessor directive.

```
#define MACRO_ID expr
```

This is very different from an `inline` function.

² <http://snet.wit.ie/GreenSpirit/c++-faq-lite/inline-functions.html>

Macro expansion provides no type checking for arguments. It is essentially an editing operation: Each occurrence of `MACRO_ID` is replaced by `expr`. Careful use of parentheses in macros is necessary to avoid precedence errors. But parentheses won't solve all the problems associated with macros, as we will see in Example 5.19. Errors caused by macros can lead to very strange (and unclear) compiler errors. Or, more dangerously, it can lead to invalid results, as the program in Example 5.19 demonstrates.

EXAMPLE 5.19 `src/functions/inlinetst.cpp`

```
// Inline functions vs macros

#include <iostream>
#define BADABS(X) (((X) < 0)? -(X) : X)
#define BADSQR(X) (X * X)
#define BADCUBE(X) (X) * (X) * (X)

using namespace std;

inline double square(double x) {
    return x * x ;
}

inline double cube(double x) {
    return x * x * x;
}

inline int absval(int n) {
    return (n >= 0) ? n : -n;
}

int main() {
    cout << "Comparing inline and #define\n" ;
    double t = 30.0;
    int i = 8, j = 8, k = 8, n = 8;
    cout << "\nBADSQR(t + 8) = " << BADSQR(t + 8)
        << "\nsquare(t + 8) = " << square(t + 8)
        << "\nBADCUBE(++i) = " << BADCUBE(++i)
        << "\ni = " << i
        << "\ncube(++j) = " << cube(++j)
        << "\nj = " << j
        << "\nBADABS(++k) = " << BADABS(++k)
        << "\nk = " << k
        << "\nabsval(++n) = " << absval(++n)
        << "\nn = " << n << endl;
}
```

Here is the output.

Comparing inline and #define

```
BADSQR(t + 8) = 278
square(t + 8) = 1444
BADCUBE(++i) = 1100
i = 11
cube(++j) = 729
j = 9
BADABS(++k) = 10
k = 10
absval(++n) = 9
n = 9
```

`BADSQR(t+8)` gives us the wrong results because

```
    BADSQR(t + 8)
=   (t + 8 * t + 8)           (preprocessor)
=   (30.0 + 8 * 30.0 + 8)     (compiler)
=   (30 + 240 + 8)           (run-time)
=   278
```

More troubling, however, are the errors produced by `BADCUBE` and `BADABS`, which both have sufficient parentheses to prevent the kind of error that occurred with `BADSQR`. Here is what happened with `BADCUBE(++i)`.

```
    BADCUBE(++i)
=   ((++i) * (++i)) * (++i) // left associativity
=   ((10) * (10)) * (11)
=   1100
```

In general, code substitution macros should be avoided. They are regarded as evil by most serious C++ programmers. Preprocessor macros are used mostly for the following:

1. `#ifndef/#define/#endif` wrapping around header files to avoid multiple inclusion
2. `#ifdef/#else/#endif` to conditionally compile some parts of code but not others
3. `__FILE__` and `__LINE__` macros for debugging and profiling

As a rule, we use `inline` functions in favor of macros for code substitutions. The exception to this rule is the use of Qt macros that insert code into programs that use certain Qt classes. It is easy to see why some C++ experts look very suspiciously at Qt's use of macros.

EXERCISES: ENCRYPTION

1. In Example 5.16, we declared but did not implement three operators for the `Point3` class. Add implementations for these three operators and add tests to the client code.
2. In this exercise, we will reuse the `random()` function from the `<cstdlib>` (see Appendix B).

`random()` generates a pseudo-random integer in the range from 0 to `RAND_MAX` (commonly set to 2147483647).

Write a function

```
int myRand(int min, int max);
```

that returns a pseudo-random `int` in the range from `min` to `max - 1`.

3. Write the function

```
QVector<int> randomPerm(int n, unsigned key);
```

that uses the `myRand()` function (seeded with `key`) to produce a permutation of the numbers `0, . . . n`.

4. Encryption and privacy are becoming increasingly important. One way to think of encryption is that we start with a string of text that we pass to one or more transforming functions. The result of these transformations is a string of encrypted text that we can then transmit more safely. The recipient of the encrypted string then applies the inverses of the transforming functions to the string of encrypted text (i.e., decrypts it) and obtains a copy of the original string. The sender of the encrypted string must share some information with the recipient that permits the string to be decrypted (i.e., a key). In the following exercises we explore a few simple designs for the transforming functions.

- a. Write the function

```
QString shift(const QString& text, unsigned key);
```

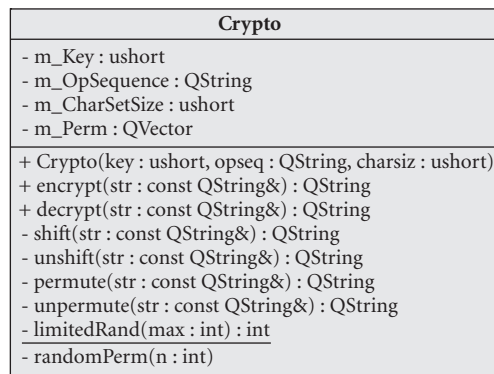
`shift()` uses the parameter `key` to set the random function's seed by calling `srandom()`. For each character `ch` in the given string, `text`, produce a shifted character by adding the next pseudo-random `int` to the code for `ch`. The shifted character is then put in the corresponding place in the new string. When all the characters of `text` have been processed, `shift()` returns the new string.

- b. The next function to write is:

```
QString unshift(const QString& cryptext, unsigned key);
```

This function reverses the process described in the previous exercise.

- c.** Write code to test the two functions described above.
- d.** Another approach to encryption (which can be combined with the approach described above) is to permute the characters of the given string. Write the function
- ```
QString permute(const QString& text, unsigned key);
```
- that uses the `randomPerm()` function to generate a permutation of the characters of the original string, `text`.
- e.** Write the function
- ```
QString unpermute(const QString& scrtext, unsigned key);
```
- that reverses the action of the `permute()` function described above.
- f.** Write code to test the two functions described above.
- g.** Write code to test `shift()` and `permute()` being applied to the same string, followed by `unpermute()` and `unshift()`.
- 5.** Implement a `Crypto` class that encapsulates the functions from the preceding exercises. You can use the following UML diagram to get you started.



`m_OpSequence` is a `QString` consisting of the characters 'p' and 's' that represent `permute()` and `shift()`. The `encrypt()` function applies those functions to the given string in the order that they appear in the `m_OpSequence` string. Example 5.20 shows some code to test your class.

Note that all of the member functions of `Crypto` are "silent" (i.e., no interaction with the user). User interactions take place only in the client code.

EXAMPLE 5.20 `src/functions/cryptoclass/crypto-client.cpp`

```
#include "crypto.h"
#include <QTextStream>
QTextStream cout(stdout, QIODevice::WriteOnly);

int main() {
    QString str1 ("This is a sample string"), str2;
    cout << "Original string: " << str1 << endl;
    QString seqstr("pspsps");
    ushort key(13579);
    Crypto crypt(key, seqstr);
    str2 = crypt.encrypt(str1);
    cout << "Encrypted string: " << str2 << endl;
    cout << "Recovered string: " << crypt.decrypt(str2) << endl;
}
```

REVIEW QUESTIONS

1. What is the difference between a function declaration and a function definition?
2. Why are default argument specifiers in the declaration but not the definition?
3. For overloading arithmetic symbols (+, -, *, /) on `Fraction` objects, which is better, member functions or non-member global operators?
4. Explain the difference between pass-by-value and pass-by-reference. Why would you use one instead of the other?
5. Explain the difference between preprocessor macros and inline functions.

6

CHAPTER 6

Inheritance and Polymorphism

This chapter introduces the concepts and shows some examples of how to define inheritance relationships between C++ classes. Overriding methods, the `virtual` keyword, and simple examples show how polymorphism works.

6.1 Simple Derivation	136
6.2 Derivation with Polymorphism	142
6.3 Derivation from an Abstract	
Base Class	148
6.4 Inheritance Design	152
6.5 Overloading, Hiding, and Overriding ...	154
6.6 Constructors, Destructors, and Copy	
Assignment Operators	155
6.7 Processing Command-Line	
Arguments	158

6.1 Simple Derivation



- Chapter 2
- Chapter 5

Inheritance is a way of organizing classes that is supported by all object-oriented languages. It allows different classes to share code in many different ways.

To employ inheritance, we place the common features of similar classes together in a **base class** and then derive other classes from it. Each **derived class** inherits all the members of the base class and can override or extend each base class function as needed. Inheritance from a common base class significantly simplifies the derived classes and, with the use of certain design patterns, allows us to eliminate redundant code.



WHAT'S WRONG WITH REPEATED CODE? Software that contains repeated pieces of identical or very similar code is error prone and difficult to maintain. If repeated code is allowed in a program, it can be difficult to keep track of all the repetitions.

Code often needs to be changed for one reason or another. If you need to change a piece of code that has been repeated several times, you must locate all of the repetitions and apply the change to each of them. Chances are good that at least one repetition will be missed or that the intended change will not be applied precisely to all repetitions.

Refactoring is a process of improving the design of software, without changing its underlying behavior. One step of refactoring involves replacing similar code with calls to library functions or base class methods.

We will demonstrate inheritance with a simple example. The base class `Student` is supposed to contain the attributes that are common to all students. We kept the list of attributes short for this example, but you can easily imagine other attributes that might be appropriate.

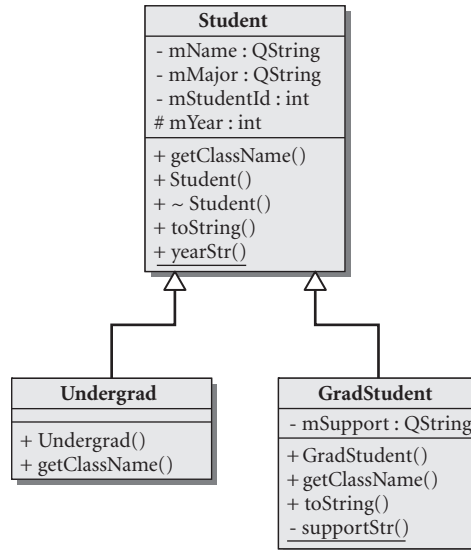


FIGURE 6.1 UML diagram of inheritance

We derived two classes from `Student` that describe particular kinds of students. The first derived class, `Undergrad`, contains only those properties that are specific to undergraduate students. The second derived class, `GradStudent`, contains only those properties that are specific to graduate students. The UML diagram shown in Figure 6.1 describes these relationships.

The pound sign (`#`) that precedes `Student : :m_Year` indicates that `m_Year` is a `protected` member of that class. Recall that `protected` members of a class are accessible to the member functions of derived classes. The open arrowhead (pointing at the base class) is used to indicate class inheritance. This arrow is also called **generalization** because it points from the more specific (derived) class to the more general (base) class. The derived classes are also called **subclasses** of the base class.

Example 6.1 shows the definitions of the three classes.

EXAMPLE 6.1 `src/derivation/qmono/student.h`

```

#ifndef STUDENT_H
#define STUDENT_H

#include <QString>

```

continued

```

class Student {
public:
    Student(QString nm, long id, QString m_Major, int year = 1);
    ~Student() {}
    QString getClassName() const; ❶
    QString toString() const;
    QString yearStr() const;
private:
    QString m_Name;
    QString m_Major;
    long m_StudentId;
protected:
    int m_Year;
};

class Undergrad: public Student {
public:
    Undergrad(QString name, long id, QString major, int year);
    QString getClassName() const;
};

class GradStudent : public Student {
public:
    enum Support { ta, ra, fellowship, other };
    GradStudent(QString nm, long id, QString major,
                int yr, Support support);

    QString getClassName() const ;
    QString toString() const;
protected:
    static QString supportStr(Support sup) ;
private:
    Support m_Support;
};

#endif // #ifndef STUDENT_H

```

- ❶ There are other ways of identifying the classname that are better than defining a `getClassName()` for each class. `getClassName()` is used here only to demonstrate how inheritance and polymorphism work.

The **classHead** of each derived class specifies the base class from which it is derived and the kind of derivation that is being used. In this case we are using `public` derivation.¹

Notice that each of the three classes has a function named `getClassName()`, and two of them have a function named `toString()`. Even though `Undergrad`

¹ We discuss the three kinds of derivation—`public`, `protected`, and `private`—in Section 23.4.

does not contain a `toString()` declaration in its definition, it inherits one from the base class.

The student functions are defined in Example 6.2.

EXAMPLE 6.2 `src/derivation/qmono/student.cpp`

```
[ . . . . ]

#include <QTextStream>
#include "student.h"

Student::Student(QString nm, long id, QString major, int year)
    : m_Name(nm), m_Major(major), m_StudentId(id), m_Year(year) {}

QString Student::getClassName() const {
    return "Student";
}

QString Student::toString() const {
    QString retval;
    QTextStream os(&retval); ❶
    os << "[" << getClassName() << "]"
        << " name: " << m_Name
        << " Id: " << m_StudentId
        << " Year: " << yearStr()
        << " Major: " << m_Major ;
    return retval;
}
```

❶ We write to the stream, and return the string it uses.

Undergrad is not very different from Student, except for one function: `getClassName()`.

EXAMPLE 6.3 `src/derivation/qmono/student.cpp`

```
[ . . . . ]

Undergrad::Undergrad(QString name, long id, QString major, int year)
    : Student(name, id, major, year) ❶
    {}

QString Undergrad::getClassName() const {
    return "Undergrad";
}
```

❶ The base class object is considered a subobject of the derived object. Class members and base classes both must be initialized and cleaned up, in an order determined by the order that they appear in the class definition.

Member Initialization for Base Classes

Because each `Undergrad` *is a* `Student`, whenever an `Undergrad` object is created, a `Student` object must also be created. In fact, one `Student` constructor is always called to initialize the `Student` part of any derived class. In the member initializers of a constructor, you can think of the base class name as an implicit member of the derived class.

- It gets initialized first, *before* the initialization of the derived class members.
- If you do not specify how the base class is initialized, the default constructor will be called.

`GradStudent` has all the features of `Student` plus some added attributes that need to be properly handled.

EXAMPLE 6.4 `src/derivation/qmono/student.cpp`

```
[ . . . ]
```

```
GradStudent::
```

```
GradStudent(QString nm, long id, QString major, int yr,
            Support support) :Student(nm, id, major, yr),
                           m_Support(support) { }
```

```
QString GradStudent::toString() const {
    QString result;
    QTextStream os(&result);
    os << Student::toString() ❶
        << "\n [Support: "    ❷
        << supportStr(m_Support)
        << " ]\n";
    return result;
}
```

❶ base class version

❷ ...plus items that are specific to `GradStudent`

Extending

Inside `GradStudent::toString()`, before the `GradStudent` attributes are printed, we explicitly call `Student::toString()`, which handles the base class attributes. In other words, `GradStudent::toString()` **extends** the functionality of `Student::toString()`.

It is worth noting here that, since most of the data members of `Student` are private, we need a base class function (e.g., `toString()`) in order to access the base class data members. A *GradStudent* object cannot directly access the

private members of Student even though it contains those members. This arrangement definitely takes some getting used to!

6.1.1 Inheritance Client Code Example

`GradStudent` *is a* `Student`, in the sense that a `GradStudent` object can be used wherever a `Student` object can be used. The client code shown in Example 6.5 creates some instances and performs operations on a `GradStudent` or an `Undergrad` instance directly and also indirectly, through pointers.

EXAMPLE 6.5 `src/derivation/qmono/student-test.cpp`

```
#include <QTextStream>
#include "student.h"

static QTextStream cout(stdout, QIODevice::WriteOnly);

void graduate(Student* student) {
    cout << "\nThe following "
         << student->getClassName()
         << " has graduated\n "
         << student->toString() << "\n";
}

int main() {
    Undergrad us("Frodo", 5562, "Ring Theory", 4);
    GradStudent gs("Bilbo", 3029, "History", 6, GradStudent::fellowship);
    cout << "Here is the data for the two students:\n";
    cout << gs.toString() << endl;
    cout << us.toString() << endl;
    cout << "\nHere is what happens when they graduate:\n";
    graduate(&us);
    graduate(&gs);
    return 0;
}
```

To build this application we use `qmake` and `make` as follows:

```
src/derivation/qmono> qmake -project
src/derivation/qmono> qmake
src/derivation/qmono> make
```

We then can run it like this:

```
src/derivation/qmono> ./qmono
Here is the data for the two students:
```

continued

```
[Student]2 name: Bilbo Id: 3029 Year: gradual student Major:
History
  [Support: fellowship]
```

```
[Student] name: Frodo Id: 5562 Year: senior Major: Ring Theory
```

Here is what happens when they graduate:

The following Student has graduated

```
[Student] name: Frodo Id: 5562 Year: senior Major: Ring Theory
```

The following Student has graduated

```
[Student] name: Bilbo Id: 3029 Year: gradual student Major:
History
src/derivation/qmono>
```

Calling `student->toString()` from the function `graduate()` invokes `Student::toString()` regardless of what kind of object `student` points to. If the object is, in fact, a `GradStudent`, then there should be a mention of the fellowship in the graduation message. In addition, we should be seeing “[`GradStudent`]” in the `toString()` messages, and we are not.

It would be more appropriate to use run-time binding for indirect function calls to determine which `toString()` is appropriate for each object.

Because of its C roots, C++ has a compiler that attempts to bind function invocations at compile time, for performance reasons. With inheritance and base class pointers, the compiler can have no way of knowing what type of object it is operating on. In the absence of run-time checking, an inappropriate function can be called. C++ requires the use of a special keyword to enable run-time binding on function calls via pointers and references. The keyword is `virtual`, and it enables polymorphism, which is explained in the next section.

6.2 Derivation with Polymorphism

We can now discuss a very powerful feature of object-oriented programming: **polymorphism**. Example 6.6 differs from the previous example in only one important way: the use of the keyword `virtual` in the base class definition.

² It would be nice if we saw [`GradStudent`] here.

EXAMPLE 6.6 `src/derivation/qpoly/student.h`

```
[ . . . . ]
class Student {
public:
    Student(QString nm, long id, QString m_Major, int year = 1);
    virtual QString getClassName() const; ❶
    QString toString() const;             ❷
    virtual ~Student() {};                ❸
    QString yearStr() const;
private:
    QString m_Name;
    QString m_Major;
    long m_StudentId;
protected:
    int m_Year;
};
// derived classes are the same as before...
[ . . . . ]
```

- ❶ Note the keyword `virtual` here.
- ❷ This should be `virtual`, too.
- ❸ It is a good idea to make the destructor `virtual`, too.

By simply adding the keyword `virtual` to at least one member function we have created a **polymorphic type**. When `virtual` is specified on a function, it becomes a method in that class and all derived classes. Example 6.7 shows the same client code again:

EXAMPLE 6.7 `src/derivation/qpoly/student-test.cpp`

```
#include <QTextStream>
#include "student.h"

static QTextStream cout(stdout, QIODevice::WriteOnly);

void graduate(Student* student) {
    cout << "\nThe following "
         << student->getClassName()
         << " has graduated\n "
         << student->toString() << "\n";
}

int main() {
    Undergrad us("Frodo", 5562, "Ring Theory", 4);
    GradStudent gs("Bilbo", 3029, "History", 6, GradStudent::fellowship);
    cout << "Here is the data for the two students:\n";
    cout << gs.toString() << endl;
    cout << us.toString() << endl;
}
```

continued

```

    cout << "\nHere is what happens when they graduate:\n";
    graduate(&us);
    graduate(&gs);
    return 0;
}

```

Running this version of the program produces slightly different output, as shown here.

Here is the data for the two students:

```

[GradStudent] name: Bilbo Id: 3029 Year: gradual student Major:
History
  [Support: fellowship ]

```

```

[Undergrad] name: Frodo Id: 5562 Year: senior Major: Ring Theory

```

Here is what happens when they graduate:

The following Undergrad has graduated

```

[Undergrad] name: Frodo Id: 5562 Year: senior Major: Ring Theory

```

The following GradStudent has graduated

```

[GradStudent] name: Bilbo Id: 3029 Year: gradual student Major:
History3

```

Notice that we now see [GradStudent] and [UnderGrad] in the output, thanks to the fact that `getClassName()` is `virtual`. There is still a problem with the output of `graduate()` for the `GradStudent`, however. The `Support` piece is missing.

With polymorphism, *indirect* calls (via pointers and references) to methods are resolved at runtime. This is sometimes called **dynamic**, or **late run-time binding**. *Direct* calls (*not* through pointers or references) of methods are still resolvable by the compiler. That is called **static binding** or **compile-time binding**.

In this example, when `graduate()` receives the address of a `GradStudent` object, `student->toString()` calls the `Student` version of the function. However, when the `Student::toString()` calls `getClassName()` (indirectly through `this`, a base class pointer), it is a `virtual` method call, bound at runtime.

Try adding the keyword `virtual` to the declaration of `toString()` in the `Student` class definition so that you can see the support data displayed properly.

In C++, dynamic binding is an option that one must switch on with the keyword `virtual`.

³ What happened to the Fellowship?

EXERCISE: DERIVATION WITH POLYMORPHISM

Be the computer and predict the output of the programs shown in Examples 6.8 through 6.12. Then compile and run the programs to check your answers.

1. EXAMPLE 6.8 src/polymorphic1.cc

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void foo() {
        cout << "A's foo()" << endl;
        bar();
    }
    virtual void bar() {
        cout << "A's bar()" << endl;
    }
};

class B: public A {
public:
    void foo() {
        cout << "B's foo()" << endl;
        A::foo();
    }
    void bar() {
        cout << "B's bar()" << endl;
    }
};

int main() {
    B bobj;
    A *aptr = &bobj;
    aptr->foo();
    A aobj = *aptr;
    aobj.foo();
}
```

2. EXAMPLE 6.9 src/polymorphic2.cc

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void foo() {
        cout << "A's foo()" << endl;
    }
};
```

continued

```

class B: public A {
public:
    void foo() {
        cout << "B's foo()" << endl;
    }
};

class C: public B {
public:
    void foo() {
        cout << "C's foo()" << endl;
    }
};

int main() {
    C cobj;
    B *bptr = &cobj;
    bptr->foo();
    A* aptr = &cobj;
    aptr->foo();
}

```

3. EXAMPLE 6.10 src/derivation/exercise/Base.h

```

[ . . . . ]
class Base {
public:
    Base();
    void a();
    virtual void b() ;
    virtual void c(bool condition=true);
    virtual ~Base() {}
};

class Derived : public Base {
public:
    Derived();
    virtual void a();
    void b();
    void c();
};
[ . . . . ]

```

EXAMPLE 6.11 src/derivation/exercise/Base.cpp

```

[ . . . . ]
Base::Base() {
    cout << "Base::Base() " << endl;
    a();
    c();
}

```

```

}
void Base::c(bool condition) {
    cout << "Base::c()" << endl;
}
void Base::a() {
    cout << "Base::A" << endl;
    b();
}
void Base::b() {
    cout << "Base::B" << endl;
}

Derived::Derived() {
    cout << "Derived::Derived() " << endl;
}

void Derived::a() {
    cout << "Derived::a()" << endl;
    c();
}
void Derived::b() {
    cout << "Derived::b()" << endl;
}

void Derived::c() {
    cout << "Derived::c()" << endl;
}
[ . . . . ]

```

EXAMPLE 6.12 src/derivation/exercise/main.cpp

```

[ . . . . ]
int main (int argc, char** argv) {

    Base b;
    Derived d;

    cout << "Objects Created" << endl;
    b.b();
    cout << "Calling derived methods" << endl;
    d.a();
    d.b();
    d.c();
    cout << ".. via base class pointers..." << endl;
    Base* bp = &d;
    bp->a();
    bp->b();
    bp->c();
}
[ . . . . ]

```

6.3 Derivation from an Abstract Base Class

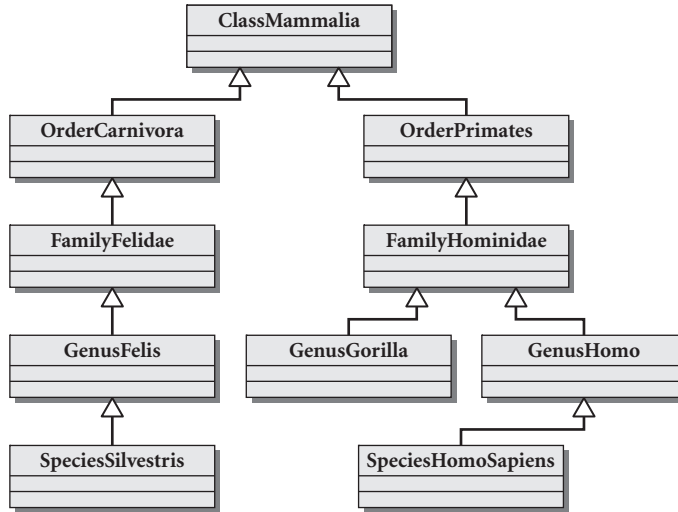


FIGURE 6.2 Animal taxonomy

An **abstract base class** is used to encapsulate common features of **concrete classes**. An abstract class cannot be instantiated. Nevertheless, this scheme is quite useful and efficient for organizing our knowledge of the vastly complex biological world. In the simplified taxonomic organization chart shown in Figure 6.2, we can see that a primate is a mammal that has certain additional characteristics, a gorilla is a hominid with certain additional characteristics, and so forth.

A concrete class represents a particular kind of entity, something that really exists and can be instantiated. For example, walking through the woods you will never encounter a real, live animal that is completely described by the designation Carnivora or Felidae. You may, depending on where you walk, find a lion, a siamese cat, or a common housecat (*Felis silvestris*). But there is no instance of a Hominidae (i.e., of a base class) in the concrete world that is not also an instance of some particular species. If a biologist ever finds a concrete instance that does not fit into an existing species definition, then that biologist may define and name a new species and become famous.

To summarize, the more general categories (class, order, family, subfamily) are all abstract base classes that cannot be instantiated in the concrete world. They

were invented by people to help with the classification and organization of the concrete classes (species).

Back to Programming

At first it might seem counterintuitive to define a class for an abstract idea that has no concrete representative. But classes are *groupings* of functions and data, and are useful tools to enable certain kinds of organization and reuse. Categorizing things makes the world simpler and more manageable for humans and computers.

As we study design patterns and develop frameworks and class libraries, we will sometimes design inheritance trees where only the leaf nodes can be instantiated, and the inner nodes are all abstract.

Once again, an abstract base class is a class that is impossible and/or inappropriate to instantiate. Features of a class that tell the compiler to enforce this rule are:

- There is at least one **pure** virtual function.
- There are no public constructors.

Figure 6.3 shows a class hierarchy with an abstract base class, `Shape`. `Shape` is abstract because it contains pure virtual functions.

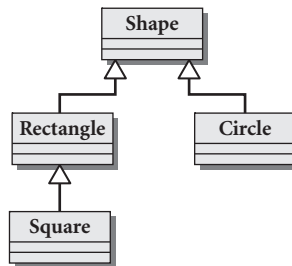


FIGURE 6.3 Shapes UML diagram

A **pure virtual function** has the following declaration syntax:

```
virtual returnType functionName(parameterList)=0;
```

Example 6.13 shows the base class definition.

EXAMPLE 6.13 `src/derivation/shape1/shapes.h`

```
[ . . . . ]

class Shape {
public:
    virtual double area() = 0;
    virtual QString getName() = 0;
```

continued

```

    virtual QString getDimensions() = 0;
    virtual ~Shape() {}
};

```

- ❶ an abstract base class
- ❷ pure virtual function

`getName()`, `area()`, and `getDimension()` are all pure virtual functions. Because they are defined to be pure virtual, no function definition is required in the `Shape` class. Any concrete derived class *must* override and define *all* pure virtual base class functions for instantiation to be permitted. In other words, any derived class that does not override and define *all* pure virtual base class functions is, itself, an abstract class. Example 6.14 shows the derived class definitions.

EXAMPLE 6.14 `src/derivation/shape1/shapes.h`

```

[ . . . . ]

class Rectangle : public Shape {
public:
    Rectangle(double h, double w) :
        m_Height(h), m_Width(w) {}
    double area();
    QString getName();
    QString getDimensions();

protected:
    double m_Height, m_Width;
};

class Square : public Rectangle {
public:
    Square(double h) : Rectangle(h,h) {}
    double area();
    QString getName();
    QString getDimensions();
};

```

- ❶ We want to access `m_Height` in `Square` class.
- ❷ Base class name in member initialization list—pass arguments to base class ctor

`Rectangle`, `Circle`, and `Square` are derived from `Shape`. Their implementations are shown in Example 6.15.

EXAMPLE 6.15 `src/derivation/shape1/shapes.cpp`

```

#include "shapes.h"

double Circle::area() {
    return(3.14159 * m_Radius * m_Radius);
}

```

```

double Rectangle::area() {
    return (m_Height * m_Width);
}
double Square::area() {
    return (Rectangle::area()); ❶
}
[ . . . . ]

```

❶ calling base class version on 'this'

We provide some client code to exercise these classes in Example 6.16.

EXAMPLE 6.16 src/derivation/shape1/shape1.cpp

```

#include "shapes.h"
#include <QString>
#include <QDebug>

void showNameAndArea(Shape* pshp) {
    qDebug() << pshp->getName()
             << " " << pshp->getDimensions()
             << " area= " << pshp->area();
}

int main() {
    Shape shp; ❶

    Rectangle rectangle(4.1, 5.2);
    Square square(5.1);
    Circle circle(6.1);

    qDebug() << "This program uses hierarchies for Shapes";
    showNameAndArea(&rectangle);
    showNameAndArea(&circle);
    showNameAndArea(&square);
    return 0;
}

```

❶ ERROR: Instantiation is not allowed on classes with pure virtual functions.

In the global function `showNameAndArea()` the base class pointer, `pshp`, is successively given the addresses of objects of the three subclasses. For each address assignment, `pshp` polymorphically invokes the correct `getName()` and `area()` functions. Example 6.17 shows the output of the program.

EXAMPLE 6.17 `src/derivation/shape1/shape.txt`

This program uses hierarchies for Shapes

```
RECTANGLE Height = 4.1 Width = 5.2 area = 21.32
CIRCLE Radius = 6.1 area = 116.899
SQUARE Height = 5.1 area = 26.01
```

6.4 Inheritance Design

Sometimes defining an inheritance relationship helps at first (e.g., by reducing redundant code), but causes problems later when other classes must be added to the hierarchy. Some up-front analysis can help make things easier and avoid problems later.

Example 6.14, in which we derived from the abstract `Shape` class, demonstrates an inheritance relationship of two levels of depth. The `Rectangle` class was used as a classification of objects and also as a concrete class.

Is a square a kind of rectangle? Geometrically it certainly is. Here are some definitions that we borrow from elementary geometry.

- A shape is a closed two-dimensional object in the plane, with a graphical way of representing itself, together with a point that is considered its “center.”
- A rectangle is a shape consisting of four straight line segments with only 90-degree angles.
- A square is a rectangle with equal sides.

As we attempt to represent an inheritance tree of classes, it helps to list the kinds of capabilities that we will need to provide for each class. They would be:

- Drawable
- Scalable
- Loadable
- Savable

After we describe the interface in further detail, the geometric definitions for shape classification may not lead to the ideal taxonomy for these shape classes.

As we perform an analysis, some questions arise:

- What are the common operations and features we want to describe in our abstract base classes?
- What other kinds of shapes will we use in our application?

- A rhombus is four-sided, like a rectangle, so should `Rectangle` derive from `Rhombus`?
- Should we have a base class for all four-sided objects?
- Is a `Square` substitutable for a `Rectangle`?
- Should we have a different base class for all five-sided objects?
- Should we have a general base class for polygons and represent the number of sides as an attribute?
- What are the common operations we will want to perform on all shapes?
- Is our program going to perform geometric proof searches to identify objects?
- Why do we need a `Rectangle` class as the base class of a `Square`?

Using a UML modeling tool makes it easier to try out different ideas before writing concrete code. UML diagrams are especially useful for focusing on and describing small parts of a larger system.

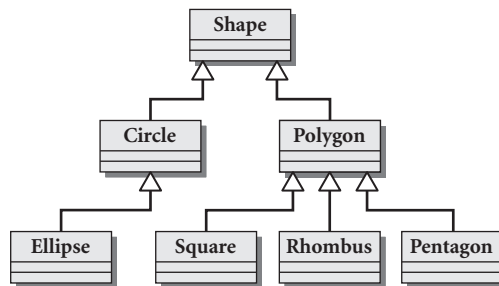


FIGURE 6.4 Another way to represent shapes

In Figure 6.4, we have concrete classes that serve as templates for creating the more “specific” shapes. The leaf classes are, in some cases, constrained versions of their base classes. The vector-representation, drawing, and loading/saving of the objects is handled in the abstract base classes.



In the geometric sense, given a circle, one can prove it is also an ellipse, because an equation exists that specifies an ellipse, with its two foci being equal. In contrast, the diagram in Figure 6.4 shows `Ellipse` to be a kind of `Circle`, with an extra point, or an extra degree of freedom. Would it make more sense to reverse the inheritance relationship? Or to have a completely different tree? Where is the *is-a* relationship?

6.5 Overloading, Hiding, and Overriding

First, let us recall the definitions of two terms that often get confused:

- When two or more versions of a function `foo` exist in the same scope (with different signatures), we say that `foo` has been **overloaded**.
- When a `virtual` function from the base class also exists in the derived class, with the *same signature*, we say that the derived version **overrides** the base class version.

Example 6.18 demonstrates overloading and overriding and introduces another relationship between functions that have the same name.

EXAMPLE 6.18 `src/derivation/overload/account.h`

```
[ . . . . ]
class Account {
protected:
    static const int SS_LEN = 80;
public:
    virtual void deposit(double amt);
    virtual const char* toString() const;
    virtual const char* toString(char delimiter); ❶
protected:
    unsigned    m_AcctNo;
    double      m_Balance;
    char        m_Owner[SS_LEN];
};

class InsecureAccount: public Account {
public:
    const char* toString() const;                ❷
    void deposit(double amt, QDate postDate); ❸
};
[ . . . . ]
```

❶ overloaded function

❷ Overrides base method and hides `toString(char)`.

❸ Does not override any method, but hides all `Account::deposit()` methods.

Function Hiding

A member function of a derived class with the same name as a function in the base class *hides* all functions in the base class with that name. In addition:

- Only the derived class function can be called directly.
- The class scope resolution operator may be used to call hidden base functions explicitly.

The client code in Example 6.19 shows how to call a base class function that has been hidden by a derived class function with the same name.

EXAMPLE 6.19 src/derivation/overload/account.cpp

```
#include "account.h"
#include <cstring>
#include <iostream>
using namespace std;

int main() {
    InsecureAccount acct;
    acct.deposit(6.23);           ❶
    acct.m_Balance += 6.23;      ❷
    acct.Account::deposit(6.23); ❸
    return 0;
}
```

- ❶ Error! No matching function—hidden by `deposit(double, int)`
- ❷ Error! Member is protected, inaccessible.
- ❸ Hidden does not mean inaccessible. We can still access hidden public members via scope resolution.

6.6 Constructors, Destructors, and Copy Assignment Operators

Three special kinds of member functions are *never* inherited:

1. Copy constructors
2. Copy assignment operators
3. Destructors

These three functions are generated automatically by the compiler for classes that do not specify them.



WHY ARE THESE FUNCTIONS SPECIAL? The base class functions are not sufficient to initialize, copy, or destroy a derived instance.

Constructors

For a class that inherits from another, the base class constructor must be called *as part of* its initialization process. The derived constructor may specify which base class constructor is called in its initialization list.

A class with no constructors is automatically given a compiler-generated default constructor that calls the default constructor for each of its base classes. If a class has some constructors but no default constructor, then it has no default initialization. In this case, any derived class constructor must make a specific base class constructor call in its initialization list.

Order of Initialization

Initialization proceeds in the following order:

1. Base classes first, in the order in which they are listed in the *classHead* of the derived class
2. Data members, in declaration order

Copy Assignment Operators

A copy assignment operator will be generated automatically by the compiler for each class that does not have one explicitly defined for it. It calls its base class `operator=` and then performs memberwise assignments in declaration order.

Other member function operators are inherited the same way as normal member functions.

Copy Constructors

Like the copy assignment operator, the copy constructor gets generated automatically for classes that do not have one defined. The compiler-generated copy constructor will carry out member-by-member initialization, much as one would expect.

EXAMPLE 6.20 `src/derivation/assigcopy/account.h`

```
[ . . . . ]  
  
class Account {  
public:  
    Account(unsigned acctNum, double balance, string owner);  
    virtual ~Account();  
private:  
    unsigned m_AcctNum;  
    double   m_Balance;  
    string   m_Owner;  
};
```

In Example 6.20, we defined a single constructor that takes arguments, so this class has no default constructor (i.e., the compiler will *not* generate one for us).

We did *not* define a copy constructor, which means the compiler *will* generate one for us. Therefore, this class can be initialized in exactly two ways.

EXAMPLE 6.21 `src/derivation/assigcopy/account.h`

```
[ . . . . ]

class JointAccount : public Account {
public:
    JointAccount (unsigned acctNum, double balance,
                  string owner, string jowner);
    JointAccount(const Account & acct, string jowner);
    ~JointAccount();
private:
    string m_JointOwner;
};
```

In the derived class defined in Example 6.21, we have two constructors. Both of them require base class initialization.

EXAMPLE 6.22 `src/derivation/assigcopy/account.cpp`

```
[ . . . . ]

#include "account.h"
#include <iostream>

Account::Account(unsigned acctNum, double balance, string owner) {
    m_Balance=balance;
    m_AcctNum = acctNum;
    m_Owner = owner;
}

JointAccount::JointAccount (unsigned acctNum, double balance,
                             string owner, string jowner)
    :Account(acctNum, balance, owner),
    m_JointOwner(jowner) { ❶

}

JointAccount::JointAccount (const Account& acc, string jowner)
    :Account(acc) { ❷
    m_JointOwner = jowner;
}

}
```

- ❶ Base class initialization is required.
 - ❷ Compiler-generated copy constructor is called.
-

In Example 6.22, the compiler allows `JointAccount::JointAccount` to call `Account(const Account&)`, even though there isn't one defined. The compiler-generated copy constructor will do a memberwise copy.

Destructors

Destructors are not inherited. Just as with the copy constructor and copy assignment operator, the compiler will generate a destructor if we do not define one explicitly. Base class destructors are automatically called on all derived objects regardless of whether a destructor is defined in the class. Members and base-class parts are destroyed in the reverse order of initialization.

6.7 Processing Command-Line Arguments

Applications that run from the command line are often controlled through command line arguments, which can be switches or parameters. `ls`, `g++`, and `qmake` are familiar examples of such applications. Handling the different kinds of command line arguments can be done in a variety of ways. Suppose that you are writing a program that supports these options:

Usage:

```
a.out [-v] [-t] inputfile.ext [additional files]
  If -v is present then verbose = true;
  If -t is present then testmode = true;
```

In usage descriptions, optional arguments are always enclosed in [square brackets] while required arguments are not. This program accepts an arbitrarily long list, consisting of at least one file name, and performs the same operation on each file.

In general, command line arguments can be any of the following:

- **Switches**, such as `-verbose` or `-t`
- **Parameters** (typically file specs), simple strings not associated with switches
- **Switched parameters** such as the gnu compiler's optional `-o` switch, which *requires* an accompanying parameter, the name of the executable file to generate

The following line contains examples of all three kinds of arguments:

```
g++ -ansi -pedantic -Wall -o myapp someclass.cpp someclass-demo.cpp
```

Example 6.23 shows how a C program might deal with command line arguments:

EXAMPLE 6.23 `src/reuse/argproc.cpp`

```
[ . . . . ]
#include <cstring>

bool test = false;
bool verbose = false;
```

```

void processFile(char* filename) {
[ . . . . ]
}
/*
  @param argc - the number of arguments
  @param argv - an array of argument strings
*/
int main (int argc, char *argv[]) {
  // recall that argv[0] holds the name of the executable.
  for (int i=1; i < argc; ++i) { ❶
    if (strcmp(argv[i], "-v")==0) {
      verbose = true;
    }
    if (strcmp(argv[i], "-t") ==0) {
      test = true;
    }
  }
  for (int i=1; i < argc; ++i) { ❷
    if (argv[i][0] != '-')
      processFile(argv[i]);
  }
}
[ . . . . ]

```

❶ first process the switches

❷ make a second pass to operate on the non-switched arguments

In Qt, we wish to avoid the use of arrays, pointers, and `<cstring>` in favor of more object-oriented constructs.

In Example 6.24, we will see how code like this could be greatly simplified through the use of higher-level classes, `QString` and `QStringList`.

6.7.1 Derivation and ArgumentList

In this section, we present the `ArgumentList` class, an example from `libutils` (see Section 7.2). It reuses `QString` and `QStringList` to simplify the processing of command-line arguments.

Operationally, `ArgumentList` is a class that we initialize with the `main()` function's `int` and `char**` parameters that capture the command line arguments. Conceptually, `ArgumentList` is a list of `QStrings`. Structurally, it is derived from `QStringList`, with some added functionality. We could also say that `ArgumentList` is *extended* from `QStringList` (as they do in Java-land).

Example 6.24 contains the class definition for `ArgumentList`.

EXAMPLE 6.24 src/libs/utls/argumentlist.h

```

#ifndef ARGUMENTLIST_H
#define ARGUMENTLIST_H

#include <QStringList>

class ArgumentList : public QStringList {
public:
    ArgumentList();

    ArgumentList(int argc, char* argv[]); {
        argsToStringlist(argc, argv);
    }

    ArgumentList(const QStringList& argumentList):
        QStringList(argumentList) {}
    bool getSwitch(QString option);
    QString getSwitchArg(QString option,
                        QString defaultRetVal=QString());
private:
    void argsToStringlist(int argc, char* argv[]);
};
#endif

```

Because it is publicly derived from `QStringList`, `ArgumentList` supports the full interface of `QStringList` and can be used wherever a `QStringList` is expected. In addition to its constructors, `ArgumentList` defines a few additional functions:

- `argsToStringlist()` extracts the command-line arguments from the given array of `char` arrays and loads them into a `QStringList`. This function is private because it is part of the implementation of this class, not part of the public interface. It is needed by the constructors but not by client code.
- `getSwitch()` finds and removes a switch from the string list, if that switch exists. It returns true if the switch was found.
- `getSwitchArg()` finds and removes a switch and its accompanying argument from the string list and returns the argument if the switch is found. It does nothing and returns a `defaultValue` if the switch is not found.

Example 6.25 shows the implementation code for these functions.

EXAMPLE 6.25 src/libs/utls/argumentlist.cpp

```

#include <QApplication>
#include "argumentlist.h"

```

```

ArgumentList::ArgumentList() {
    if (qApp != NULL) ❶
        argsToStringlist(qApp->argc(), qApp->argv());
}

void ArgumentList::argsToStringlist(int argc, char * argv []) {
    for (int i=0; i < argc; ++i) {
        *this += argv[i];
    }
}

bool ArgumentList::getSwitch (QString option) {
    QMutableStringListIterator itr(*this);
    while (itr.hasNext()) {
        if (option == itr.next()) {
            itr.remove();
            return true;
        }
    }
    return false;
}

QString ArgumentList::getSwitchArg(QString option, QString
defaultValue) {
    if (isEmpty())
        return defaultValue;
    QMutableStringListIterator itr(*this);
    while (itr.hasNext()) {
        if (option == itr.next()) {
            itr.remove();
            QString retval = itr.next();
            itr.remove();
            return retval;
        }
    }
    return defaultValue;
}

```

❶ a global pointer to the current `qApplication`

In the client code shown in Example 6.26, all argument processing code has been removed from `main()`. No loops, `char*`, or `strcmp` are to be found.

EXAMPLE 6.26 `src/reuse/main.cpp`

```

#include <QString>
#include <QDebug>
#include <argumentlist.h> // in our utils lib

void processFile(QString filename, bool verbose) {
    if (verbose)

```

continued

```

        qDebug() << QString("Do something chatty with
        %1.").arg(filename);
    else
        qDebug() << filename;
}

void runTestOnly(QStringList & listOfFiles, bool verbose) {
    foreach (QString current, listOfFiles) { ❶
        processFile(current, verbose);
    }
}

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

    ArgumentList al(argc, argv);                ❷
    QString appname = al.takeFirst();           ❸
    qDebug() << "Running " << appname;

    bool verbose = al.getSwitch("-v");
    bool testing = al.getSwitch("-t");          ❹
    if (testing) {
        runTestOnly(al, verbose);              ❺
        return 0;
    } else {
        qDebug() << "This Is Not A Test";
    }
}

```

- ❶ Qt improved foreach loop.
- ❷ Instantiate the ArgumentList with command line args.
- ❸ The first item in the list is the name of the executable.
- ❹ Now all switches have been removed from the list. Only filenames remain.
- ❺ ArgumentList can be used in place of QStringList.

Here are some sample outputs from running the program in Example 6.26:

```

src/reuse> ./reuse item1 "item2 item3" item4 item5
"Running ./reuse"
This Is Not A Test
src/reuse> ./reuse -t item1 "item2 item3" item4 item5
"Running ./reuse"
"item1"
"item2 item3"
"item4"
"item5"
src/reuse> ./reuse -v -t "foo bar" 123 space1 "1 1"
"Running ./reuse"
"Do something chatty with foo bar."
"Do something chatty with 123."
"Do something chatty with space1."
"Do something chatty with 1 1."
src/reuse>

```

The project file in Example 6.27 shows how to reuse the classes in our `utils` library.

EXAMPLE 6.27 `src/reuse/reuse.pro`

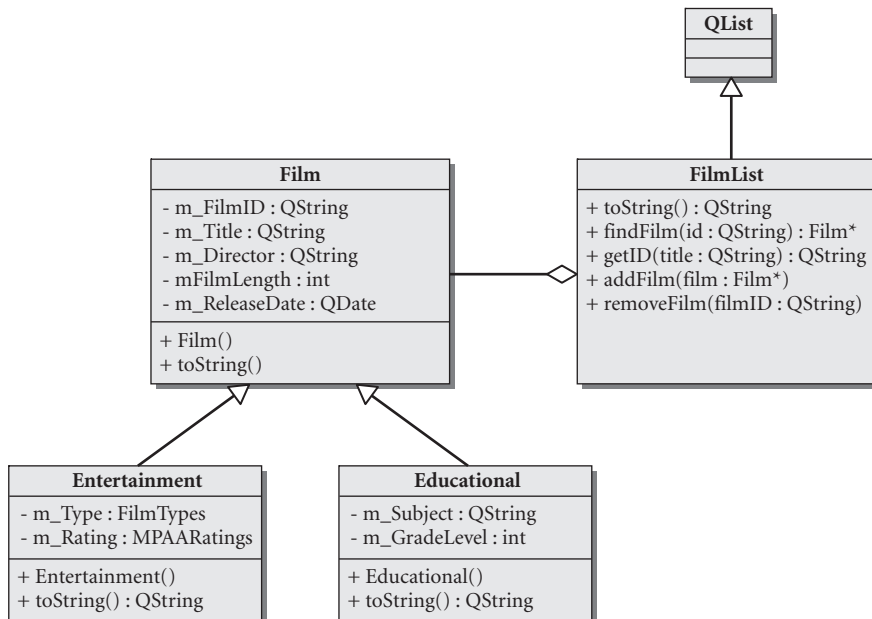
```

TEMPLATE = app
INCLUDEPATH += $$ (CPPLIBS) /utils
LIBS += -L$$ (CPPLIBS) -lutils
# Input
SOURCES += main.cpp

```

EXERCISE: INHERITANCE AND POLYMORPHISM

1. The classes in the diagram below are intended to help organize the film collection in the college library.
 - a. Implement the `Film` classes. Make sure that the constructors have sufficient parameters to initialize all data members. We have suggested enum types `FilmTypes` (Action, Comedy, SciFi,...) and `MPAARatings` (G, PG, PG-13,...) for use in the `Entertainment` class.



- b.** Implement the `FilmList` class as a container of `Film` pointers. Make sure that the `addFilm()` function does not permit the same `Film` to be added more than once.
 - c.** Write client code to test these classes. Put a mixture of `Entertainment` and `Educational` films into the `FilmList` and exercise all the member functions.
- 2.** Revisit the birthday reminder exercise (“Exercise: The Qt Core Module” in Chapter 3) and use an `ArgumentList` object to process the command-line arguments.

POINTS OF DEPARTURE

1. Which methods are in `QStringList` but not in `QList`? Would it make sense to put them in `QList`?
2. Examine the Qt class hierarchy.⁴ The most important classes are those with the most derived classes. Which three classes have the most derived classes? Write a paragraph about each of them and why you think it is important.

⁴ <http://oop.mcs.suffolk.edu/qtdocs/hierarchy.html>

REVIEW QUESTIONS

1. What is inheritance for?
2. Explain polymorphism. What is it? How can you use it?
3. Explain the difference between dynamic binding and static binding. Describe the conditions that enable each.
4. How do you override a base class method?
5. What is a pure virtual function? What is it used for?
6. What is an abstract class? What is it used for? What can you do with a concrete class that you *cannot* do with an abstract class?
7. What does it mean for a base class function to be *hidden*? What can cause this to happen?
8. Which member functions *cannot* be inherited from the base class? Explain why.



P A R T I I

Higher-Level Programming

Chapter 7. Libraries	169
Chapter 8. Introduction to Design Patterns..	181
Chapter 9. QObject	191
Chapter 10. Generics and Containers	213
Chapter 11. Qt GUI Widgets	237
Chapter 12. Concurrency.....	277
Chapter 13. Validation and Regular Expressions	307
Chapter 14. Parsing XML.....	321
Chapter 15. Meta Objects, Properties, and Reflective Programming....	341
Chapter 16. More Design Patterns	359
Chapter 17. Models and Views.....	391
Chapter 18. Qt SQL Classes.....	423


7

CHAPTER 7

Libraries

Libraries are groups of code modules organized in a reusable way. This chapter introduces how they are built, reused, and designed.

7.1 Code Containers	170
7.2 Reusing Other Libraries	171
7.3 Organizing Libraries: Dependency Management	173
7.4 Installing Libraries: A Lab Exercise	176
7.5 Frameworks and Components	178



Libraries generally contain code that has already been designed, tested, and compiled, and can be easily linked into your application. Libraries are essential for making software reuse possible. They can be packaged in a number of different ways, such as

1. Source code
2. Binary format (dynamic library, shared object, static library, run-time library), called **lib** for short.
3. lib + header files (sometimes referred to as “-dev” or “-devel” packages in Linux package managers)



The lib+header combo allows you to distribute your library without the full source. Others can still compile their apps with it. The binary format can only be used with an app that was already compiled against the library.

A lib is a file that contains several compiled files (called **object files**) that are indexed to make it easy for the linker to locate symbols (e.g., names of classes, members, functions, variables, etc.) and their definitions. Packaging these object files in one lib expedites the linking process significantly.

7.1 Code Containers

One aspect of C++ that makes it very powerful is its ability to package code in several different ways.

Table 7.1 defines some terms that are used to describe containers of code. The table is arranged (approximately) in order of increasing granularity from top to bottom.

TABLE 7.1 Reusable Components

Term	Visible attributes	Description
class	<code>class Classname</code> <code>{ body };</code>	A collection of functions and data members, and descriptions of its lifecycle management (constructors and destructors)
namespace	<code>namespace name</code> <code>{ body };</code>	A collection of declarations and definitions, of classes, functions and static members, perhaps spanning multiple files
header file	<code>.h</code>	Class definitions, template definitions, function declarations (with default argument definitions), inline definitions, static object declarations
source code module	<code>.cpp</code>	Function definitions, static object definitions
compiled "object" module	<code>.o</code> or <code>.obj</code>	Each <code>.cpp</code> module is compiled into a binary module as an intermediate step in building a library or executable.
library	<code>.lib</code> or <code>.la</code> (+ <code>.so</code> or <code>.dll</code> if dynamic)	An indexed collection of object files linked together. No <code>main()</code> function must exist in any code module in a library.
devel package	lib + header files	A library along with accompanying header files
application	<code>.exe</code> on windows, no particular extension on *nix	A collection of object files, linked with libraries, to form an application. Contains exactly one function definition called <code>main()</code> .

7.2 Reusing Other Libraries

Many of our examples link with various libraries that we have supplied. The one we use most frequently is called `libutils`. You can download a tarball containing this library here.¹ Create a shell/environment variable `CPPLIBS` that points to a convenient (empty) directory and then unpack the `utils` tarball in that directory.

¹ <http://oop.mcs.suffolk.edu/dist>



When we set up projects that reuse `utils` we will always assume that the shell/environment variable `$CPPLIBS` (or `%CPPLIBS%` in Windows) has been properly set to contain the “libs root.” This variable is used for two purposes: It is the parent directory of all the C++ source code for libraries supplied by us (or by you), and it is also the destination directory of the compiled shared object code for those libraries.

`qmake` can access an environment variable such as `CPPLIBS` from inside a project file using the syntax `$$ (CPPLIBS)`. `qmake` can also include other project file (fragments). For example, the project file in Example 7.1 includes a `.pri` file located in a directory relative to your environment variable `CPPLIBS`.

EXAMPLE 7.1 `src/qapp-gui/qapp-gui.pro`

```
include ($$(CPPLIBS)/utils/common.pri)

TEMPLATE = app
# Input
HEADERS += messenger.h
SOURCES += main.cpp messenger.cpp
```

The command

```
qmake -project
```

produces a project file that contains information based on the contents of the current working directory. In particular, `qmake` cannot know about external libraries that you may need to build your project. So, if your project depends on an external library, you must edit the project file and add assignments to three of the variables.

For example, suppose we are developing an application that uses our `utils` library. The header files are located in `$CPPLIBS/utils` and the lib shared object files are located in `$CPPLIBS`. Then we must add the following lines to the project file

```
INCLUDEPATH += $$ (CPPLIBS)/utils # the source header files
LIBS += -L$$ (CPPLIBS)           # add this to the lib search path
LIBS += -lutils                  # link with libutils.so
```

Assignments to the `LIBS` variable generally contain two kinds of linker switches that are passed directly to the compiler and the linker. For more information about what the linker switches mean see Section C.2.

7.3 Organizing Libraries: Dependency Management

A **dependency** between two program elements exists if one is reusing the other, that is, if using or testing one (the reuser) requires the presence and correctness of the other one (the reused). In the case of classes, a dependency exists if the implementation of the reuser class must be changed whenever the interface of the reused class is changed.

Another way of describing this relationship is to say that `ProgElement1` *depends on* `ProgElement2` if `ProgElement2` is needed in order to build `ProgElement1`.

This dependency is a **compile-time dependency** if `ProgElement1.h` must be `#included` in `ProgElement2.cpp` in order to compile.

It is a **link-time dependency** if the object file `ProgElement2.o` contains symbols that are defined in `ProgElement1.o`.

We depict the dependency between a reuser `ClassA` and a reused `ClassB` with a UML diagram, as shown in Figure 7.1.

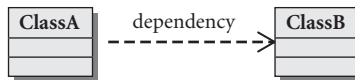


FIGURE 7.1 Dependency

A dependency between `ClassA` and `ClassB` can arise in a variety of ways. In each of the following situations, a change in the interface of `ClassB` might necessitate changes in the implementation of `ClassA`.

- `ClassA` has a data member that is a `ClassB` object or pointer.
- `ClassA` is derived from `ClassB`.
- `ClassA` has a function that takes a parameter of type `ClassB`.
- `ClassA` has a function that uses a static member of `ClassB`.
- `ClassA` sends a message (e.g., a signal) to `ClassB`.²

In each case, it is necessary to `#include ClassB` in the implementation file for `ClassA`.

In the **package diagram** shown in Figure 7.2, we have displayed parts of our own `libs` collection of libraries. There are direct and indirect dependencies shown. At this level of granularity we are concerned with the *dependencies between libraries* (indicated by dashed arrows).

² We discuss signals and slots in Section 9.3.2.

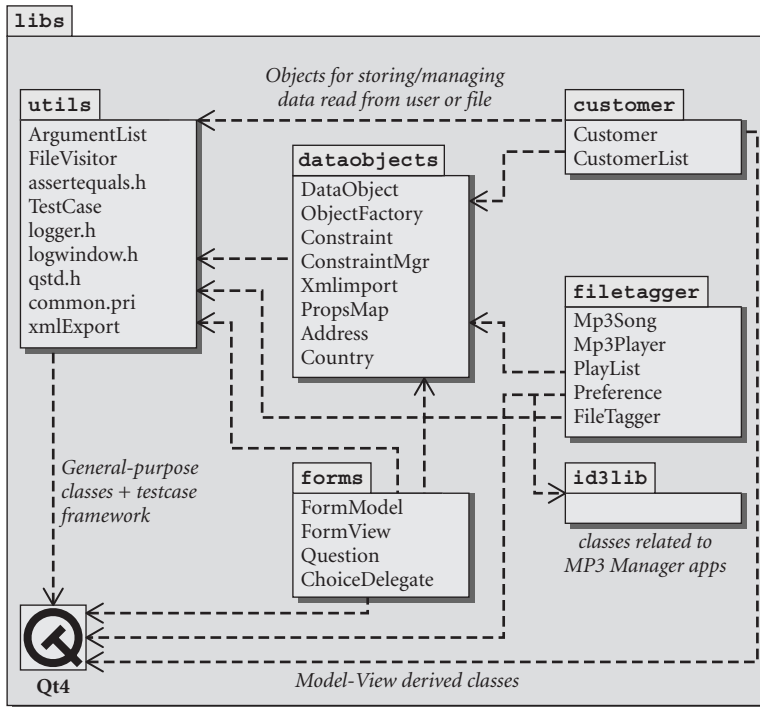


FIGURE 7.2 Libraries and their dependencies

If you wish to reuse one of the libraries shown in Figure 7.2, you need to ensure that all of its dependent libraries are also part of your project. For example, if you use the `filetagger` library, there is a chain of dependencies that requires you to also make available the `dataobjects` library (e.g., `FileTagger` is derived from `DataObject`), the `utils` library (e.g., the test code generally reuses the various assert macros in `utils`), and the `id3lib` library (e.g., `FileTagger` has a member of type `auto_ptr<ID3_Tag*>` and defines some functions with parameters of type `ID3_FrameID*`).

Code reuse, a valuable and important goal, *always* produces dependencies. When designing classes and libraries it is important to make sure that we produce as few unnecessary or unintentional dependencies as possible because they tend to slow down compile times and reduce the reusability of your classes and libraries. Each `#include` directive produces a dependency and should be carefully examined to make sure that it is really necessary. This is especially true in header files: Each time a header file is `#included` it brings all of its own `#includes` along with it so that the number of dependencies grows accordingly.



A **forward declaration** of a class declares its name as a valid class name but leaves out its definition. This permits that name to be used as a type for pointers and references that are not dereferenced before the definition is encountered. Forward declarations make it possible for classes to have circular relationships without having circular dependencies between header files (which the compiler will not permit).

In a class definition, one good rule to follow is this: Do not use an `#include` if a *forward declaration* will suffice. For example, the header file "classa.h" might look something like this:

```
#include "classb.h"
#include "classd.h"
// other #include directives as needed
class ClassC;    // forward declaration
class ClassA : public ClassB {
    public:
        ClassC* f1(ClassD);
        // other stuff that does not involve ClassC
};
```

We have (at least) two intentional reuse dependencies in this definition: `ClassB` and `ClassD`, so both `#include` directives are necessary. A forward declaration of `ClassC` is sufficient, however, since the class definition only involves a pointer to that class.

It is very important to make sure that there are no circular dependencies in your project (i.e., in a diagram like the one in Figure 7.2, there must be no path that permits you to return to the starting location by following a sequence of arrows.)³

Dependency management is an important issue that is the subject of several articles and for which a variety of tools have been developed. Two open-source tools are

- `cinclude2dot`, a perl script that analyzes C/C++ code and produces a dependency graph.
- `Makedep`, a C/C++ dependency generator for large software projects that parses all source files in a directory tree and constructs a large dependency file for inclusion in a Makefile.

³ Such a path is called a **cycle**.

7.3.1 Installing Libraries

After a library has been written and tested, it will be installed at the end of the build process in the directory specified by the `DESTDIR` variable. For example, the project file for our `utils` library contains the following relevant lines:

```
TEMPLATE = lib          # Build this as a library, not as an application
DESTDIR=${CPPLIBS}     # Place the compiled shared object code here
```

For library templates, `qmake` can generate a Makefile with the `install` target so that the command

```
make install
```

will, after a successful build, copy the library to some particular location. For example, on a *nix platform, the following lines could be added to the project file for `utils`:

```
target.path=/usr/lib
INSTALLS += target
```

Then, if you have write access there, the command

```
make install
```

would copy the `libutils.so` files and their associated symlinks to the directory `/usr/lib`.

If you need to relocate a library, the procedure varies from platform to platform. In Windows, you can copy its `.dll` file into an appropriate directory that is listed in your `PATH` variable. In *nix, you can copy the shared object file and associated symbolic links into a directory that is listed in `/etc/ld.so.conf` or one that is findable by searching in `LD_LIBRARY_PATH`.

During development, it is usually sufficient to make and install libraries in your home directory, and adjust `LD_LIBRARY_PATH` appropriately. At deployment time, on a *nix platform, it may be desirable to install the library in `/usr/local`, a systemwide location accessible to all other users. This would require superuser permissions.

7.4 Installing Libraries: A Lab Exercise

A number of examples in this book make use of classes found in one of the libraries that were written for this book. The source code for these classes is available for download.⁴ In the HTML version of this book, the class names are all

⁴ <http://oop.mcs.suffolk.edu/dist>

hyperlinked to API docs.⁵ Here you learn how to download and install these extra classes.

Instructions for installing libraries on a *nix platform for use with the book examples follow.

- Create a `~/projects/libs` directory.
- Create a shell/environment variable `CPPLIBS` that contains the absolute path of `projects/libs`.
- You should already have a shell/environment variable `QTDIR` that contains the absolute path of the directory that contains Qt 4.⁶
- Download `libs.pro`, `utils.tar.gz`, and `dataobjects.tar.gz`.⁷
- Unpack these two tarballs in the `$CPPLIBS` directory. That should result in two new subdirectories, `$CPPLIBS/utils` and `$CPPLIBS/dataobjects`.
- Create or modify a `$CPPLIBS/libs.pro` SUBDIRS project to build only `libutils` followed by `libdataobjects`.
- Build the libraries from the `libs` directory in two steps:
 1. `qmake //` to create a Makefile
 2. `make //` to build the library
- Verify that the libraries are built and that the shared object files (e.g., `libutils.so`) are located in the `$CPPLIBS` directory.

7.4.1 Fixing the Linker Path

- Update the shell/environment variable `LD_LIBRARY_PATH` (*nix) or `PATH` (win32) to include `CPPLIBS`.
- Create a `projects/tests` directory. This is where you can keep code for testing various library components.
- Download `main.cpp` from Example 6.26 into `projects/tests` and write/modify a `tests.pro` file to build that application using the `utils` library.
- Run the application.
- Verify that it gives the same output that we displayed back in Chapter 6 when we discussed the `ArgumentList` class.

⁵ <http://oop.mcs.suffolk.edu/api>

⁶ This is not required by Qt 4, but is required by Qt 3 and KDE, and is used for Qt 4 as a way of pointing to “the current version of Qt that you are using.”

⁷ From <http://oop.mcs.suffolk.edu/dist>



On a *nix platform a shell script is generally used to define environment variables. Example 7.2 shows a bash script that handles the job.

EXAMPLE 7.2 src/bash/env-script.sh

```
export CPPLIBS=$HOME/cs331/projects/libs
export QTDIR=/usr/local/qt
export LD_LIBRARY_PATH=$QTDIR/lib:$CPPLIBS
export PATH=$QTDIR/bin:$PATH
```

You can run this script by typing one of the following commands:

```
source env-script.sh
```

or

```
. env-script.sh
```

Notice the dot (.) at the beginning of the second version. In the bash shell, the dot is equivalent to the command, `source`.

If you want to make sure that these environment variables are automatically set at the start of each shell, you can source the script from `~/ .bashrc`, which runs automatically whenever `bash` starts.

Hamish Whittal has put together a very nice online guide to shell scripting at <http://learnlinux.tsf.org.za/courses/build/shell-scripting>.

7.5 Frameworks and Components

Organization of classes goes beyond simple inheritance. Carefully designed frameworks allow one to find and reuse components much more easily. All large software projects are built on top of frameworks, and we discuss some of the more popular ones in use today.

Code reuse is the holy grail of programming. In the past, computer time was expensive and programmer time was relatively cheap, but now things are exactly reversed. Today all software is built out of building blocks, which are themselves pieces of software. We never start from scratch. It is a waste of programmers' time to reinvent and reimplement things that have already been designed, implemented, refined, and tested by recognized experts.

A **framework** is a (typically large) collection of general-purpose (or domain-specific) classes and conventions designed to improve the consistency of design. Frameworks are often used to create graphical applications, database applications, or other complex pieces of software.

A framework has a well-documented public Application Programmers' Interface, or **API**. An API is a description of the public functions, classes, and interfaces in a library. To implement frameworks, **design patterns** are used. Development with design patterns involves looking for pertinent objects and possible hierarchies. The classes and patterns used are given good descriptive names so that we can define them once and reuse them elsewhere. We will discuss design patterns shortly in Chapter 8.

Qt 4 is one of many open-source object-oriented frameworks that provide a set of reusable components for building cross-platform applications. Some others worth knowing about are:

- **boost**: an open-source cross-platform library of C++ utility classes
- **mono**: an open-source implementation of Microsoft's .NET, the API for C#, which is built on top of libgtk
- **libgtk, libgtk++**: the widgets used under the Gnome desktop, Mozilla, GAIM, GIMP, Evolution, and many other open-source programs
- **wxWidgets**: another C++ cross platform widget toolkit
- **Wt**⁸: a Qt-like framework for building Web applications using boost and AJAX.⁹

With a multi-platform framework like Qt 4, you can gain enormous benefits from the creative efforts of others. Software built *on top of* (strictly using) Qt 4 will be based on components that have already been tested on Windows, Linux, and Mac OS/X by hundreds of programmers.

Toolkits like Qt 4 (and also Gtk++, the cross-platform Gnu Toolkit) have parts that are implemented differently on each platform. This is why Qt-based applications look like KDE apps in Linux and like Windows apps in Windows.

⁸ <http://jose.med.kuleuven.ac.be/wt>

⁹ A system of JavaScript and XML-rpc that gives list/tree/table views inside a Web page

REVIEW QUESTIONS

1. For each of these items, list whether it would be found in a header (.h) file or an implementation (.cpp) file, and why.
 - a. Function definitions
 - b. Function declarations
 - c. Static object declarations
 - d. Static object definitions
 - e. Class definitions
 - f. Class declarations
 - g. Inline function definitions
 - h. Inline function declarations
 - i. Default argument specifiers
2. What is the difference between a compile-time dependency and a link-time dependency?


8

CHAPTER 8

Introduction to Design Patterns

Design patterns are efficient and elegant solutions to common problems in object-oriented software design. They are high-level abstract templates that can be applied to particular kinds of design problems.

8.1 Iteration and the Visitor Pattern 182



In their very influential book, *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often (affectionately) referred to as the “Gang of Four,” analyzed 23 specific patterns [Gamma95]. Each pattern has a section devoted to it, including

- A pattern name
- A description of the kinds of problems to which one might apply the pattern
- An abstract description of a design problem and how its solution can be obtained
- A discussion of the results and trade-offs that can occur when the pattern is applied

Design patterns are used for many different purposes. As a result, they are subdivided into categories. The three main categories are

- **Creational patterns**, which manage the creation of objects
- **Structural patterns**, which describe how objects are connected together to form more complex objects
- **Behavioral patterns**, which are used to describe how code is organized, to assign responsibility or roles to certain classes, and to specify the way objects communicate with each other

The Gang of Four assert that design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” As we continue to develop applications with Qt, we will discuss and give examples of several design patterns.

8.1 Iteration and the Visitor Pattern

In this section, we discuss some elegant ways to reuse `QFile`, `QDir`, and `QFileInfo`. We present a visitor object that processes each file in a file system. By separating the visitor code from the application logic, we have a `FileVisitor` that can be used by other programs. Maximum code reuse is demonstrated through careful interface design.

8.1.1 Directories and Files: QDir and QFileInfo

There will be many occasions when it is necessary to write a program to visit files of a certain type, and to perform some operation on each. Since directories can contain subdirectories as well as files, the visiting code may need to be recursive. Qt has two particularly useful classes for visiting files and getting information about their contents: `QDir` and `QFileInfo`. Using these classes, we can write portable (Win32, MacOS, and *nix) code that searches a file system.

Example 8.1 will assemble a list of absolute path names of MP3 files that are scattered throughout the subdirectories below a given path. The final product might be a playlist that we present to a piece of software such as Winamp, XMMS, or amaroK. Here is our first try.

EXAMPLE 8.1 `src/visitorsrc/recurseaddir.cpp`

```

/* This function searches for all Mp3 files in a directory
   tree and adds them. */

void Mp3Db::recurseAddDir(QDir d) {
    d.setSorting( QDir::Name );
    d.setFilter( QDir::Files | QDir::Dirs );
    QStringList qsl = d.entryList();
    foreach (QString entry, qsl) {
        QFileInfo finfo(entry);
        if ( finfo.isSymLink () && !m_SymLink )
            return;
        if ( finfo.isDir() ) {
            if (!m_Recursive )
                return;
            QString dirname = finfo.fileName();
            if ((dirname=="." ) || (dirname == ".."))
                return;
            QDir d(finfo.filePath());
            if (skipDir(d))
                return;
            recurseAddDir(d);
        } else {
            if (finfo->extension(false)=="mp3") {
                addMp3File(finfo->absFilePath()); ❶
            }
        }
    }
}

```

❶ non-reusable part

Example 8.1 shows how the APIs of `QDir` and `QFileInfo` can be used for traversing directories. It is not well-designed for reuse, however, since there is application-specific code in the example.

8.1.2 Visitor Pattern

The **Visitor pattern**, like most design patterns, describes how to separate code in a more reusable way. In this case, an operation is to be performed on selected objects in some organized collection. The idea of this pattern is to maintain a separation between visiting code and the code that processes each visited item. Then we can plug application-specific code for processing each selected file into our reusable visiting code.

In this section, we present some code from our `libutils` (see Section 7.4) called `FileVisitor`. Given a path to a directory, `FileVisitor` traverses all the files and subdirectories in that path. We can impose constraints so that we skip over files that we do not wish to process. `FileVisitor` applies the function `processFile()` to each file that satisfies our constraints.

To make `FileVisitor` into a reusable tool, we keep the file-processing operation separate from the file-traversing code. There are two ways we can plug in the processing code: through inheritance/polymorphism or signals/slots (see Section 9.3.3).

To use the inheritance approach, we override the virtual method `processFile()`. By deriving from `FileVisitor`, we can customize its behavior as it deals with individual files. Since the `FileVisitor`'s `processFile` emits a signal, we could connect the signal to a slot and avoid inheritance entirely.

EXAMPLE 8.2 `src/libs/utils/filevisitor.h`

```
[ . . . . ]
#include <QDir>
#include <QObject>

class FileVisitor : public QObject {
    Q_OBJECT
public:
    FileVisitor(QString nameFilter="",
               bool recursive=true, bool symlinks=false);
public slots:
    void processFileList(QStringList sl);
    void processEntry(QString pathname);
```

```

signals:
    void foundFile(QString filename);
protected:
    virtual void processFile(QString filename);

    [ . . . . ]
protected:
    QString m_NameFilter;
    bool m_Recursive;
    QDir::Filters m_DirFilter;
};
[ . . . . ]

```

The constraints (`m_NameFilter`, `m_DirFilter`, and `m_Recursive`) are stored as data members of `FileVisitor`, shown in Example 8.2. If `m_Recursive` is true, then the `FileVisitor` will descend into each subdirectory that it encounters. Example 8.3 shows how the visitor deals with files and directories. Notice, that since the `FileVisitor`'s `processFile` emits a signal, we could create a concrete `FileVisitor` and connect the `foundFile()` to a slot, avoiding inheritance entirely.

EXAMPLE 8.3 `src/libs/utils/filevisitor.cpp`

```

[ . . . . ]
void FileVisitor::processFile(QString filename) { ❶
    // qDebug() << QString("FileVisitor::processFile(%1)").arg(filename);
    emit foundFile(filename);
}

void FileVisitor::processEntry(QString current) {
    QFileInfo finfo(current);
    processEntry(finfo);
}

void FileVisitor::processEntry(QFileInfo finfo) {
    // qDebug(QString("ProcessEntry: %1").arg(finfo.fileName()));

    if (finfo.isDir()) {
        QString dirname = finfo.fileName();
        if ((dirname==".") || (dirname==".."))
            return;
        QDir d(finfo.filePath());
        if (skipDir(d))
            return;
        processDir(d);
    } else
        processFile(finfo.filePath());
}

```

continued

```
[ . . . . ]
void FileVisitor::processDir(QDir& d) {
    QStringList filters;
    filters += m_NameFilter;
    d.setSorting( QDir::Name );
    QStringList files = d.entryList(filters, m_DirFilter);
    foreach(QString entry, files) {
        processEntry(d.filePath(entry));
    }

    if (m_Recursive) {
        QStringList dirs = d.entryList(QDir::Dirs);
        foreach(QString dir, dirs) {
            processEntry(d.filePath(dir));
        }
    }
}
[ . . . . ]
```

❶ Override this method to do something more interesting with each file.

After all the files in a directory's list are processed and if `m_Recursive` is true, `processDir()` obtains a list of entries and calls `processEntry()` recursively on each of them.

8.1.3 Customizing the Visitor Using Inheritance

Figure 8.1 shows an extended `FileVisitor`, customized for the specific purpose of tracking code dependencies.

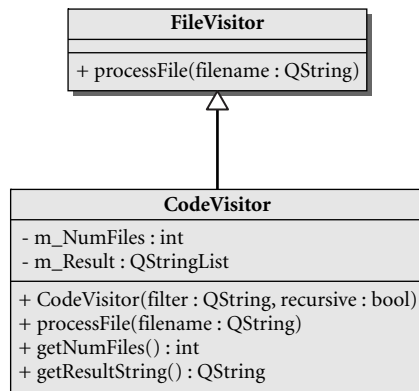


FIGURE 8.1 Code Visitor UML

The `codevisitor` application opens each file that it visits and collects all the `#include` lines in a `QStringList`. The class definition is shown in Example 8.4.

EXAMPLE 8.4 `src/visitor/codevisitor/codevisitor.h`

```
[ . . . . ]
class CodeVisitor : public FileVisitor {
public:
    CodeVisitor(QString filter = "-h", bool recursive = true) :
        FileVisitor(filter, recursive), m_NumFiles(0) {}
    void processFile(QString filename);
    int getNumFiles() const;
    QString getResultString() const;
private:
    int m_NumFiles;
    QStringList m_Result;
};
[ . . . . ]
```

The implementation shown in Example 8.5, consists mainly of overriding the `processFile()` method.

EXAMPLE 8.5 `src/visitor/codevisitor/codevisitor.cpp`

```
[ . . . . ]

void CodeVisitor::processFile(QString filename) {
    m_Result << filename;
    QString line;
    QFile file(filename);
    file.open(QIODevice::ReadOnly);
    QTextStream filestream(&file);
    while(! filestream.atEnd()) {
        line = filestream.readLine();
        if(line.startsWith("#include"))
            m_Result << QString(" %1").arg(line);
    }
    file.close();
    ++m_NumFiles;
}
}
```

Example 8.6 is an application for reporting on dependencies of source code files.

EXAMPLE 8.6 `src/visitor/codevisitor/codevisitor-test.cpp`

```
#include <argumentlist.h>
#include <codevisitor.h>
#include <QApplication>
#include <qstd.h>
using namespace qstd;
```

continued

```

void usage(QString appname) {
    cout << appname << " Usage: \n"
         << "codevisitor [-r] [-d startdir] [-f filter] [file-
            list]\n"
         << "\t-r          \tvisitor will recurse into subdirs\n"
         << "\t-d startdir\tspecifies starting directory\n"
         << "\t-f filter\tfilename filter to restrict visits\n"
         << "\toptional list of files to be visited" << endl;
}

int main(int argc, char** argv) {
    ArgumentList al(argc, argv);
    QString appname = al.takeFirst(); ❶
    if(al.count() == 0) {
        usage(appname);
        exit(1);
    }
    bool recursive(al.getSwitch("-r"));
    QString startdir(al.getSwitchArg("-d"));
    QString filter(al.getSwitchArg("-f"));
    CodeVisitor cvis(filter, recursive);
    if(startdir != QString()) {
        cvis.processEntry(startdir);
    }
    else if(al.size()) {
        cvis.processFileList(al);
    }
    else
        return 1;
    cout << "Files Processed: " << cvis.getNumFiles() << endl;
    cout << cvis.getResultString() << endl;
    return 0;
}

```

❶ app name is always first in the list.

Decoupling

Notice that `codevisitor` also reuses our `ArgumentList` class. `main()` passes an `ArgumentList` to the `FileVisitor` function, `processFileList` (`QStringList`). Both `FileVisitor` and `ArgumentList` depend on `QStringList`, but neither depends on the other. Thus, each can be compiled and reused separately. We can also pass one as an argument to the other's function.

Now, `main()` is free of iteration code, conditionals, and data structure manipulations. Dealing with these higher-level objects makes `main()` easier to read and understand. Here are some sample runs of this app.

```

src/visitor/codevisitor> ./codevisitor
./codevisitor Usage:
codevisitor [-r] [-d startdir] [-f filter] [file-list]

```



```

        -r                visitor will recurse into subdirs
        -d startdir      specifies starting directory
        -f filter        filename filter to restrict visits
                        optional list of files to be visited
src/visitor/codevisitor> ./codevisitor -r -d "../" -f "*.h"
Files Processed: 3
../imagemport.h
    #include <filevisitor.h>
    #include <QApplication>
    #include <qconsole.h>
../qconsole.h
    #include <QMainWindow>
../codevisitor/codevisitor.h
    #include <filevisitor.h>
    #include <QStringList>
src/visitor/codevisitor>

```

EXERCISES: ITERATION AND THE VISITOR PATTERN

1. `diskusage` reports, in various formats, the amount of disk space used in a directory tree. It traverses all files and computes total file sizes, providing different levels of detail (depending on command line arguments) in its report.

```

diskusage [-v] [-b] [-k] [-m] fileOrDirName
          [fileName2 dirName3 ...]

```

Option for <code>diskusage</code>	Description
<code>-v</code>	Verbose—show each file visited and its total
<code>-k</code>	Print size in kilobytes (1024 bytes)
<code>-m</code>	Print size in megabytes (1024 kbytes)

- a. Write a tool `diskusage` that prints the summary information (totals) of files found for each argument supplied (directory or file), in kilobytes. For example:


```

OOP> diskusage .
30102k  .

```
 - b. Implement the options `-v`, `-k`, and `-m`. Check that subdirectories in the traversal each have their own totals properly calculated. Compare your results to the *nix command `du`.
2. Use `FileVisitor` to implement the playlist problem that was described in Section 8.1.1.

REVIEW QUESTIONS

1. What is a design pattern? What is an example of a design pattern? Why would you use it?
2. Why does the `FileVisitor` need to be recursive?
3. There are three kinds of design patterns: structural, creational, and behavioral. Which kind is the visitor? Why?
4. `FileVisitor` could be used to make changes to selected files. Testing such an app would be quite risky. How would you do it?


9

CHAPTER 9

QObject

An important class to become familiar with is the one from which all Qt Widgets are derived: `QObject`.

9.1 QObject's Child Management	194
9.2 Composite Pattern: Parents and Children	196
9.3 QApplication and the Event Loop	200
9.4 Q_OBJECT and moc: A Checklist	209
9.5 Values and Objects	210
9.6 tr() and Internationalization	211



We will refer to any object of a class derived from `QObject` as a `QObject`. Here is an abbreviated look at its definition.

```
class QObject {
public:
    QObject(QObject* parent=0);
    QObject * parent () const;
    QString objectName() const;
    void setParent ( QObject * parent );
    const ObjectList & children () const;
    // ... more ...
};
```

The first interesting thing that we observe is that `QObject`'s copy constructor is not public. `QObject`s are not meant to be copied. In general, `QObject`s are intended to represent unique objects with identity; that is, they correspond to real-world things that also have some sort of persistent identity. One immediate consequence of not having access to its copy constructor is that a `QObject` can never be passed by value to any function. Copying a `QObject`'s data members into another `QObject` is still possible, but the two objects are still considered unique.

One immediate consequence of not having access to its copy constructor is that `QObject`s can never be passed by value to any function.

Each `QObject` can have (at most) one **parent** object and an arbitrarily large container of `QObject*` children. Each `QObject` stores pointers to its children in a `QObjectList`.¹ The list itself is created in a lazy-fashion to minimize the overhead for objects which do not use it. Since each child is a `QObject` and can have an arbitrarily large collection of children, it is easy to see why copying `QObject`s is not permitted.

The notion of children can help to clarify the notion of identity and the no-copy policy for `QObject`s. If you represent individual humans as `QObject`s, the idea of a unique identity for each `QObject` is clear. Also clear is the idea of children. The rule that allows each `QObject` to have at most one parent can be seen as a way to simplify the implementation of this class. Finally, the no-copy policy stands out as a clear necessity. Even if it were possible to “clone” a person (i.e., copy

¹ `QObjectList` is a typedef (i.e., an alias) for `QList<QObject*>`.

its data members to another `QObject`), the question of what to do with the children of that person makes it clear that the clone would be a separate and distinct object with a different identity.

Each `QObject` parent *manages* its children. This means that the `QObject` destructor automatically destroys all of its child objects.

The child list establishes a **bidirectional**, one-to-many association between objects. Setting the parent of one object implicitly adds its address to the child list of the other, for example

```
objA->setParent(objB);
```

adds the `objA` pointer to the child list of `objB`. If we subsequently have

```
objA->setParent(objC);
```

then the `objA` pointer is removed from the child list of `objB` and added to the child list of `objC`. We call such an action **reparenting**.

Parent Objects versus Base Classes

Parent objects should not be confused with **base classes**. The parent-child relationship is meant to describe containment, or management, of objects at runtime. The base-derived relationship is a static relationship between classes determined at compile-time.

It is possible that a parent can also be an instance of a base class of some of its child objects. These two kinds of relationships are distinct and must not be confused, especially considering that many of our classes will be derived directly or indirectly from `QObject`.

It is already possible to understand some of the reasons for not permitting `QObject`s to be copied. For example, should the copy have the same parent as the original? Should the copy have (in some sense) the children of the original? A shallow copy of the child list would not work because then each of the children would have two parents. Furthermore, if the copy gets destroyed (e.g., if the copy was a value parameter in a function call), each child needs to be destroyed too. Even with resource sharing methods, this approach would introduce some serious difficulties. A deep copy of the child list could be a costly operation if the number of children were large and the objects pointed to were large. Since each child could also have arbitrarily many children, this questionable approach would also generate serious difficulties.

9.1 QObject's Child Management

Example 9.1 shows a QObject derived class.

EXAMPLE 9.1 src/qobject/person.h

```
[ . . . . ]
class Person : public QObject {

public:
    Person(QObject* parent, QString name);
    virtual ~Person();
};
[ . . . . ]
```

The complete implementation is shown in Example 9.2 to show that there is no explicit object deletion done in ~Person().

EXAMPLE 9.2 src/qobject/person.cpp

```
#include "person.h"
#include <QTextStream>
static QTextStream cout(stdout, QIODevice::WriteOnly);

Person::Person(QObject* parent, QString name)
    : QObject(parent) {
    setObjectName(name);
    cout << QString("Constructing Person: %1").arg(name) << endl;
}

Person::~Person() {
    cout << QString("Destroying Person: %1").arg(objectName()) <<
    endl;
}
```

main(), shown in Example 9.3, creates some objects, adds them to other objects, and then exits. All heap objects were implicitly destroyed.

EXAMPLE 9.3 src/qobject/main.cpp

```
#include <QTextStream>
#include "person.h"

static QTextStream cout(stdout, QIODevice::WriteOnly);
```

```

int main(int , char**) {
    cout << "First we create a bunch of objects." << endl;
    Person bunch(0, "A Stack Object"); ❶
    /* other objects are created on the heap */
    Person *mike = new Person(&bunch, "Mike");
    Person *carol = new Person(&bunch, "Carol");
    new Person(mike, "Greg");           ❷
    new Person(mike, "Peter");
    new Person(mike, "Bobby");
    new Person(carol, "Marcia");
    new Person(carol, "Jan");
    new Person(carol, "Cindy");
    new Person(0, "Alice");           ❸
    cout << "\nDisplay the list using QObject::dumpObjectTree()"
         << endl;
    bunch.dumpObjectTree();
    cout << "\nProgram finished - destroy all objects." << endl;
    return 0;
}

```

- ❶ not a pointer
- ❷ We do not need to remember pointers to children, since we can reach them via object navigation.
- ❸ Alice has no parent—memory leak?

Here is the output of this program:

```

First we create a bunch of objects.
Constructing Person: A Stack Object
Constructing Person: Mike
Constructing Person: Carol
Constructing Person: Greg
Constructing Person: Peter
Constructing Person: Bobby
Constructing Person: Marcia
Constructing Person: Jan
Constructing Person: Cindy
Constructing Person: Alice

Display the list using QObject::dumpObjectTree()
QObject::A Stack Object
  QObject::Mike
    QObject::Greg
    QObject::Peter
    QObject::Bobby
  QObject::Carol
    QObject::Marcia
    QObject::Jan
    QObject::Cindy

```

continued

```

Program finished - destroy all objects.
Destroying Person: A Stack Object
Destroying Person: Mike
Destroying Person: Greg
Destroying Person: Peter
Destroying Person: Bobby
Destroying Person: Carol
Destroying Person: Marcia
Destroying Person: Jan
Destroying Person: Cindy

```

Notice that Alice is not part of the `dumpObjectTree()` and does not get destroyed.

EXERCISE: QOBJECT'S CHILD MANAGEMENT

Add the function

```
void showTree(QObject* theparent)
```

to `main.cpp`. The output of this function, after all objects have been created, should look like this:

```

Member: Mike - Parent: A Stack Object
Member: Greg - Parent: Mike
Member: Peter - Parent: Mike
Member: Bobby - Parent: Mike
Member: Carol - Parent: A Stack Object
Member: Marcia - Parent: Carol
Member: Jan - Parent: Carol
Member: Cindy - Parent: Carol

```

9.2 Composite Pattern: Parents and Children

According to [Gamma95], the **Composite pattern** is intended to facilitate building complex (composite) objects from simpler (component) parts by representing the part-whole hierarchies as tree-like structures. This must be done in such a way that clients do not need to distinguish between simple parts and more complex parts that are made up of (i.e., contain) simpler parts.

In Figure 9.1 there are two distinct classes for describing the two roles.

- A **composite object** is something that can contain children.
- A **component object** is something that can have a parent.

In Figure 9.2, we can see that `QObject` is both composite *and* component. We can express the whole-part relationship as a parent-child relationship between `QObject`s. The highest level (i.e., most “composite”) `QObject` in such a tree

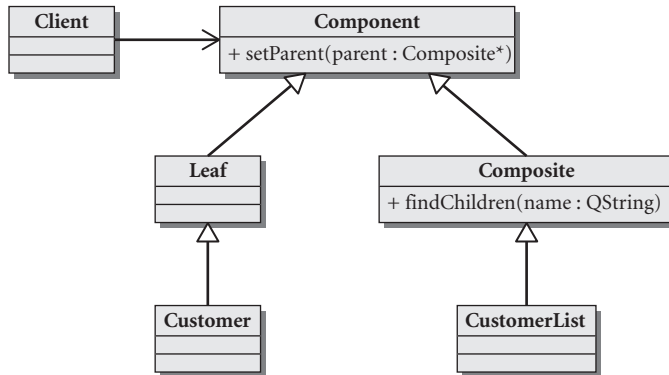


FIGURE 9.1 Components and composites

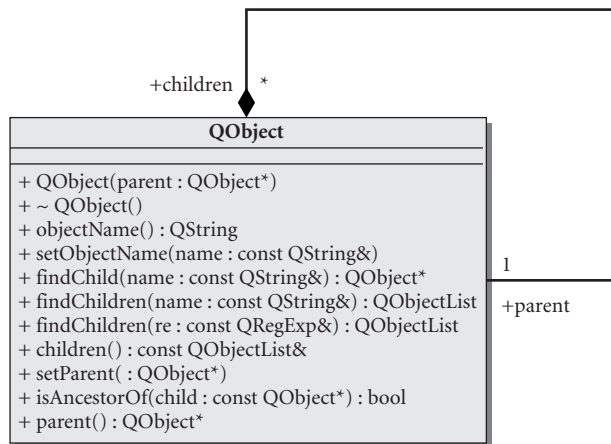


FIGURE 9.2 QObject

(i.e., the **root** of the tree) will have lots of children but no parent. The simplest `QObject`s (i.e., the **leaf nodes** of this tree) will each have a parent but no children. Client code can recursively deal with each node of the tree.

For an example of how this pattern might be used, let's look at Suffolk University. In 1906 the founder, Gleason Archer, decided to start teaching the principles of law to a small group of tradesmen who wanted to become lawyers. He was assisted by one secretary and, after a while, a few instructors. The organizational chart for this new school was quite simple: a single office consisting of several employees with various tasks. As the enterprise grew, the chart gradually became more complex with the addition of new offices and departments. Today, 100 years

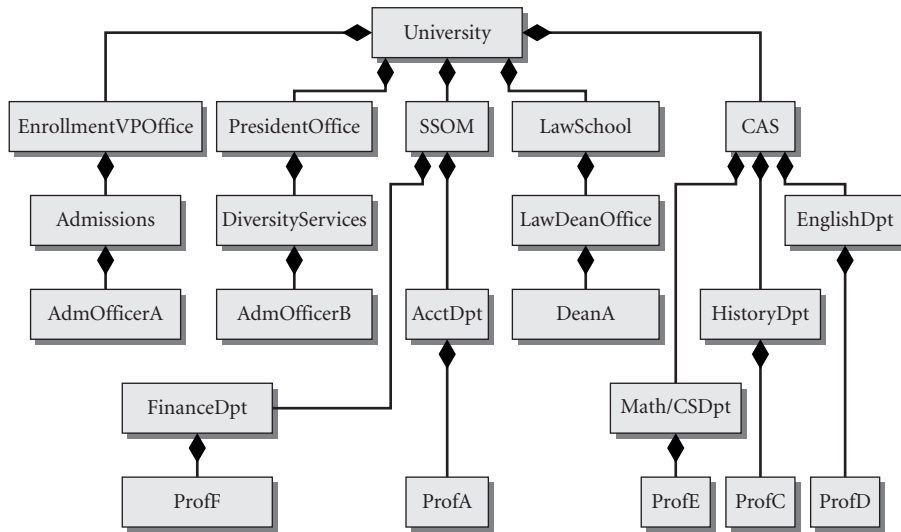


FIGURE 9.3 Suffolk University organizational chart

later, the Law School has been joined with a College of Arts and Sciences, a School of Management, a School of Art and Design, campuses abroad, and many specialized offices so that the organizational chart has become quite complex and promises to become more so. Figure 9.3 shows an abbreviated and simplified subchart of today's Suffolk University.

Each box in the chart is a component. It may be composite and have sub-components which, in turn, may be composite or simple components. For example, the PresidentOffice has individual employees (e.g., the President and his assistants) and sub-offices (e.g., DiversityServices). The leaves of this tree are the individual employees of the organization.

We can use the Composite pattern to model this structure. Each node of the tree can be represented by an object of

```

class OrgUnit : public QObject {
public:
    QString getName();
    double getSalary();
private:
    QString m_Name;
    double m_Salary;
};

```

The `QObject` public interface allows us to build up a tree-like representation of the organization with code that instantiates an `OrgUnit` and then calls `setParent()` to add it to the appropriate child list.

For each `OrgUnit` pointer `ouptr` in the tree, we initialize its `m_Salary` data member as follows:

- If `ouptr` points to an individual employee, we use that employee's actual salary.
- Otherwise we initialize it to 0.

We can implement the `getSalary()` method somewhat like this:

```
double OrgUnit::getSalary() {
    QList<OrgUnit*> childlst = findChildren<OrgUnit*>();
    double salaryTotal(m_Salary);
    if(!childlst.isEmpty())
        foreach(OrgUnit* ouptr, childlst)
            salaryTotal += ouptr->getSalary();
    return salaryTotal;
}
```

A call to `getSalary()` from any particular node returns the total salary for the part of the university represented by the subtree whose root is that node. For example, if `ouptr` points to `University`, `ouptr->getSalary()` returns the total salary for the entire university. But if `ouptr` points to `EnglishDpt`, then `ouptr->getSalary()` returns the total salary for the English Department.

9.2.1 Finding Children

`QObject` provides convenient and powerful functions named `findChildren()` for finding children in the child list. The signature of one of its overloaded forms looks like this:

```
QList<T> parentObj.findChildren<T> ( const QString & name ) const
```

If `name` is an empty string, `findChildren()` works as a class filter by returning a `QList` holding pointers to all children, which can be typecast to type `T`.

To call the function, you must supply a template parameter after the function name, as shown in Example 9.4.

EXAMPLE 9.4 src/findchildren/findchildren.cpp

```
[ . . . . ]
/* Filter on Customer* */
    QList<Customer*> custlist = parent.findChildren<Customer*>();
    foreach (Customer* current, custlist) {
        qDebug() << current->toString();
    }
[ . . . . ]
```

9.3 QApplication and the Event Loop

Interactive Qt applications with GUI have a different control flow from console applications and filter applications² because they are event-based, and often multi-threaded. Objects are frequently sending messages to each other, making a linear hand-trace through the code rather difficult.

Observer Pattern

When writing event-driven programs, GUI views need to respond to changes in the state of data model objects, so that they can display the most recent information possible.

When a particular subject object changes state, it needs an *indirect* way to alert (and perhaps send additional information to) all the other objects that are listening to state-change events, known as observers. A design pattern that enables such a message-passing mechanism is called the **Observer pattern**, sometimes also known as the **Publish-Subscribe pattern**.

There are many different implementations of this pattern. Some common characteristics that tie them together are

1. They all enable concrete subject classes to be decoupled from concrete observer classes.
2. They all support broadcast-style (one to many) communication.
3. The mechanism used to send information from subjects to observers is completely specified in the subject's base class.

Qt's approach is very different from Java's approach, because signals and slots rely on generated code, while Java just renames observer to listener.

The Qt class `QEvent` encapsulates the notion of an event. `QEvent` is the base class for several specific event classes such as `QActionEvent`, `QFileOpenEvent`, `QHoverEvent`, `QInputEvent`, `QMouseEvent`, and so forth. `QEvent` objects can be created by the window system in response to actions of the user (e.g., `QMouseEvent`) at specified time intervals (`QTimerEvent`) or explicitly by an application program. The `type()` member function returns an `enum` that has nearly a hundred specific values that can identify the particular kind of event.

A typical Qt program creates objects, connects them, and then tells the application to `exec()`. At that point, the objects can send information to each other in a

² A filter application is not interactive. It simply reads from standard input and writes to standard output.

variety of ways. `QWidget`s send `QEvents` to other `QObject`s in response to user actions such as mouse clicks and keyboard events. A widget can also respond to events from the window manager such as repaints, resizes, or close events. Furthermore, `QObject`s can transmit information to one another by means of signals and slots.

Each `QWidget` can be specialized to handle keyboard and mouse events in its own way. Some widgets will emit a signal in response to receiving an event.

An **event loop** is a program structure that permits events to be prioritized, enqueued, and dispatched to objects. Writing an event-based application means implementing a **passive interface** of functions that only get called in response to certain events. The event loop generally continues running until a terminating event occurs (e.g., the user clicks on the QUIT button).

Example 9.5 shows a simple application that initiates the event loop by calling `exec()`.

EXAMPLE 9.5 `src/eventloop/main.cpp`

```
[ . . . . ]

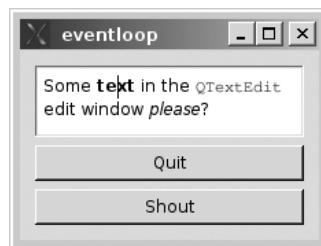
int main(int argc, char * argv[]) {
    QApplication myapp(argc, argv); ❶

    QWidget rootWidget;
    setGui(&rootWidget);

    rootWidget.show();               ❷
    return myapp.exec();             ❸
};
```

- ❶ Every GUI, multithreaded, or event-driven Qt Application must have a `QApplication` object defined at the top of `main()`.
- ❷ Show our widget on the screen.
- ❸ Enter the event loop.

When we run this app, we first see a widget on the screen as shown in the following figure.



We can type in the `QTextEdit` on the screen, or click on the Shout button. When Shout is clicked, a widget is superimposed on our original widget as shown in the next figure.



This message dialog knows how to self-destruct, because it has its own buttons and actions.

9.3.1 Layouts: A First Look

Whenever more than a single widget needs to be displayed, they must be arranged in some form of a **layout** (see Section 11.5). Layouts are derived from the abstract base class, `QLayout`, which is derived from `QObject`. Layouts are geometry managers that fit into the composition hierarchy of a graphical interface. Typically, we start with a widget that will contain all of the parts of our graphical construction. We select one or more suitable layouts to be children of our main widget (or of one another) and then we add widgets to the layouts.



It is important to understand that widgets in a layout are not children of the layout—they are children of the widget that owns the layout. Only a widget can be the parent of another widget. It may be useful to think of the layout as an older sibling acting as the nanny of its widgets.

In Example 9.6, we are laying out widgets in a vertical fashion with `QVBoxLayout`.

EXAMPLE 9.6 `src/eventloop/main.cpp`

```
[ . . . . ]

QWidget* setGui(QWidget *box) {
    QLayout* layout = new QVBoxLayout;
    box->setLayout(layout); 1

    QTextEdit *te = new QTextEdit; 2
    layout->addWidget(te); 3
}
```

```

te->setHtml("Some <b>text</b> in the <tt>QTextEdit</tt>"
        "edit window <i>please</i>?");

QPushButton *quitButton=new QPushButton("Quit");
layout->addWidget(quitButton);

QPushButton *shoutButton = new QPushButton("Shout");
layout->addWidget(shoutButton);

Messenger *msgr = new Messenger("This dialog will self-
deconstruct.", box);

QObject::connect(quitButton, SIGNAL(clicked()),
                 qApp, SLOT(quit())); ❹

qApp->connect(shoutButton, SIGNAL(clicked()), msgr, SLOT(shout()));
return box;
}

```

- ❶ box is the parent of layout.
- ❷ This is the window for qDebug messages.
- ❸ te is the child of layout.
- ❹ qApp is a global variable that points to the current QApplication object.

The widgets are arranged vertically in this layout, from top to bottom, in the order that they were added to the layout.

9.3.2 Connecting to Slots

In Example 9.7, we saw the following connections established:

```

QObject::connect(quitButton, SIGNAL(clicked()), qApp, SLOT(quit()));
qApp->connect(shoutButton, SIGNAL(clicked()), msgr, SLOT(shout()));

```

`connect()` is actually a static member of `QObject` and can be called with any `QObject` or, as we showed, by means of its class scope resolution operator. `qApp` is a global pointer that points to the currently running `QApplication`.

The second `connect` goes to a `slot` that we declare in Example 9.7.

EXAMPLE 9.7 `src/eventloop/messenger.h`

```

#ifndef MESSENGER_H
#define MESSENGER_H

#include <QObject>
#include <QString>
#include <QErrorMessage>

```

continued

```
class Messenger : public QObject {
    Q_OBJECT
public:
    Messenger (QString msg, QWidget* parent=0);

public slots:
    void shout();

private:
    QWidget* m_Parent;
    QErrorMessage* message;
};

#endif
```

Declaring a member function to be a `slot` enables it to be connected to a `signal` so that it can be called passively in response to some event. For its definition, shown in Example 9.8, we have kept things quite simple: the `shout()` function simply pops up a message box on the screen.

EXAMPLE 9.8 `src/eventloop/messenger.cpp`

```
#include "messenger.h"

Messenger::Messenger(QString msg, QWidget* parent)
    : m_Parent(parent) {
    message = new QErrorMessage(parent);
    setObjectName(msg);
}

void Messenger::shout() {
    message->showMessage(objectName());
}
```

9.3.3 Signals and Slots

When the main thread of a C++ program calls `qApp->exec()`, it enters into an event loop, where messages are handled and dispatched. While `qApp` is executing its event loop, it is possible for `QObject`s to send messages to one another.

A **signal** is a message that is presented in a class definition like a `void` function declaration. It has a parameter list but no function body. A signal is part of the interface of a class. It looks like a function but it cannot be called—it must be *emitted* by an object of that class. A signal is implicitly `protected`, and so are all the identifiers that follow it in the class definition until another access specifier appears.

A **slot** is a `void` member function. It can be called as a normal member function.

A signal of one object can be *connected* to the slots of one or more³ other objects, provided the objects exist and the parameter lists are assignment compatible⁴ from the signal to the slot. The syntax of the connect statement is:

```
bool QObject::connect(senderqobjptr,
                      SIGNAL(signalname(argtypelist)),
                      receiverqobjptr,
                      SLOT(slotname(argtypelist))
                      optionalConnectionType);
```

Any `QObject` that has a signal can emit that signal. This will result in an indirect call to all connected slots.

`QWidget`s already emit signals in response to events, so you only need to make the proper connections to receive those signals. Arguments passed in the emit statement are accessible as parameters in the slot function, similar to a function call, except that the call is indirect. The argument list is a way to transmit information from one object to another.

Example 9.9 defines a class that uses signals and slots to transmit a single `int` parameter.

EXAMPLE 9.9 `src/widgets/sliderlcd/sliderlcd.h`

```
[ . . . . ]
class QSlider;
class QLCDNumber;
class LogWindow;
class QErrorMessage;

class SliderLCD : public QMainWindow {
    Q_OBJECT
public:
    SliderLCD(int minval = -273, int maxval = 360);
    void initSliderLCD();

public slots:
    void checkValue(int newValue);
    void showMessage();

signals:
    void toomuch();
private:
    int m_Minval, m_Maxval;
    LogWindow* m_LogWin;
    QErrorMessage *m_ErrorMessage;
```

continued

³ Multiple signals can be connected to the same slot also.

⁴ Same number of parameters, each one being assignment compatible.

```

    QLCDNumber* m_LCD;
    QSlider* m_Slider;
};
#endif
[ . . . . ]

```

In Example 9.10, we can see how the widgets are initially created and connected.

EXAMPLE 9.10 `src/widgets/sliderlcd/sliderlcd.cpp`

```

[ . . . . ]

SliderLCD::SliderLCD(int min, int max) : m_Minval(min),
m_Maxval(max) {
    initSliderLCD();
}

void SliderLCD::initSliderLCD() {
    m_LogWin = new LogWindow();
    QDockWidget *logDock = new QDockWidget("Debug Log");
    logDock->setWidget(m_LogWin);
    logDock->setFeatures(0);
    setCentralWidget(logDock);

    m_LCD = new QLCDNumber();
    m_LCD->setSegmentStyle(QLCDNumber::Filled);
    QDockWidget *lcdDock = new QDockWidget("LCD");
    lcdDock->setFeatures(QDockWidget::DockWidgetClosable);
    lcdDock->setWidget(m_LCD);
    addDockWidget(Qt::LeftDockWidgetArea, lcdDock);

    m_Slider = new QSlider( Qt::Horizontal);
    QDockWidget* sliderDock = new QDockWidget("How cold is it
today?");
    sliderDock->setWidget(m_Slider);
    sliderDock->setFeatures(QDockWidget::DockWidgetMovable);
    /* Can be moved between doc areas */
    addDockWidget(Qt::BottomDockWidgetArea, sliderDock);

    m_Slider->setRange(m_Minval, m_Maxval);
    m_Slider->setValue(0);
    m_Slider->setFocusPolicy(Qt::StrongFocus);
    m_Slider->setSingleStep(1);
    m_Slider->setPageStep(20);
    m_Slider->setFocus();

    connect(m_Slider, SIGNAL(valueChanged(int)), /*SliderLCD is a
QObject so
        connect does not need scope resolution. */
        this, SLOT(checkValue(int)));

```

```

connect(m_Slider, SIGNAL(valueChanged(int)),
        m_LCD, SLOT(display(int)));

connect(this, SIGNAL(toomuch()),
        this, SLOT(showMessage()));
m_ErrorMessage = NULL;
}

```

⑦

- ① a class defined in the utils library
- ② cannot be closed, moved, or floated
- ③ can be closed
- ④ Step each time left or right arrow key is pressed.
- ⑤ Step each time PageUp/PageDown key is pressed.
- ⑥ Give the slider focus.
- ⑦ Normally there is no point in connecting a signal to a slot on the same object, but we do it for demonstration purposes.

Only the argument types belong in the `connect` statement; for example, the following is not legal:

```
connect( button, SIGNAL(valueChanged(int)), lcd, SLOT(setValue(3)))
```

Example 9.11 defines the two slots, one of which conditionally emits another signal.

EXAMPLE 9.11 `src/widgets/sliderlcd/sliderlcd.cpp`

```

[ . . . . ]

void SliderLCD::checkValue(int newValue) {
    if (newValue > 120) {
        emit tooMuch();
    }
}

/* This slot is called indirectly via emit because
   of the connect */
void SliderLCD::showMessage() {
    if (m_ErrorMessage == NULL) {
        m_ErrorMessage = new QErrorMessage(this);
    }
    if (!m_ErrorMessage->isVisible()) {
        QString message("Too hot outside! Stay in. ");
        m_ErrorMessage->showMessage(message);
    }
}

```

①

②

- ① Emit a signal to anyone interested.
- ② This is a direct call to a slot. It's a member function.

Example 9.12 contains client code to test this class.

EXAMPLE 9.12 `src/widgets/sliderlcd/sliderlcd-demo.cpp`

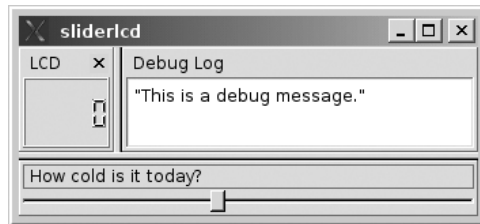
```

#include "sliderlcd.h"
#include <QApplication>
#include <QDebug>

int main(int argc, char ** argv) {
    QApplication app(argc, argv);
    SliderLCD slcd;
    slcd.show();
    qDebug() << QString("This is a debug message.");
    return app.exec();
}

```

Whenever the `slider` produces a new value, that value is transmitted as an argument from the `valueChanged(int)` signal to the `display(int)` slot of the `lcd`.

**Synchronous or Asynchronous?**

In single-threaded applications, or in multithreaded applications where the emitting and receiving `QObject`s are in the same thread, signals are sent in a **synchronous** manner. This means the thread blocks (suspends execution) until the code for the slots has completed execution (see Section 12.2).

In multi-threaded applications, where signals are emitted by an object in one thread and received by an object in another, it is possible to have signals queued, or executed in an **asynchronous** way, depending on the optional `Qt::ConnectionType` passed to `connect()`.

EXERCISES: SIGNALS AND SLOTS

1. Modify the `sliderlcd` program as follows:
 - Make the `lcd` display show the temperatures as hexadecimal integers.
 - Make the `lcd` display characters have a different ("flat") style.

- Give the slider a vertical orientation.
 - Give the slider and the lcd display more interesting colors.
 - Add a push button that the user can click to switch the lcd display from decimal mode to hexadecimal mode.
 - Make the push button into a toggle that allows the user to switch back and forth between decimal and hexadecimal modes.
2. Write an application, similar to the one in Section 9.3, but that has four buttons. The first one, labeled *Advice*, should be connected to a slot that randomly selects a piece of text (such as a fortune cookie) and displays it in the `QTextEdit` window. The second one, labeled *Weather*, randomly selects a sentence about the weather and displays it in the `QTextEdit` window. The third one, labeled *Next Meeting*, pops up a message dialog with a randomly generated (fictitious) meeting time and descriptive message in it. The fourth one, labeled *Quit*, terminates the program. Use signals and slots to connect the button clicks with the appropriate functions.

9.4 Q_OBJECT and moc: A Checklist

`QObject` supports features not normally available in C++ objects.

- Children (see the first two sections of Chapter 9)
- Signals and slots (see Section 9.3.3)
- MetaObjects, metaproperties, metamethods (see Chapter 15)
- `qobject_cast` (see Section 15.3)

These features are only possible through the use of generated code. The Meta Object Compiler, `moc`, generates additional functions for each `QObject`-derived class that uses the macro. Generated code can be found in files with names `moc_filename.cpp`.

This means that some errors from the compiler/linker may be confuscated⁵ when `moc` is not able to find or process your classes. To help ensure that `moc` processes each of your `QObject`-derived classes, here are some guidelines for writing C++ code and `qmake` project files.

- Each class definition should go in its own `.h` file.
- Its implementation should go in a corresponding `.cpp` file.
- The header file should be “`#ifndef` wrapped” to avoid multiple inclusion.
- Each source (`.cpp`) file should be listed in the `SOURCES` variable of the project file, otherwise it will not be compiled.

⁵ confusing + obfuscated

- The header file should be listed in the `HEADERS` variable of the `.pro` file. Without this, `moc` will not preprocess the file.
- The `Q_OBJECT` macro must appear inside the class definition, so that `moc` will know to generate code for it.



MULTIPLE INHERITANCE AND QOBJECT Because each `Q_OBJECT` has signals and slots, it needs to be preprocessed by `moc`. `moc` works under the assumption that you are only deriving from `QObject` once, and further, that it is the first base class in the list of base classes. If you accidentally inherit from `QObject` multiple times, or if it is not the first base class in the inheritance list, you may receive very strange errors from `moc`-generated code.

9.5 Values and Objects

We can divide C++ types into two categories: value types and object types.

Instances of **value types** are relatively “simple”: They occupy contiguous memory space, and can be copied or compared quickly. Examples of value types are `Anything*`, `int`, `char`, `QString`, `QDate`, and `QVariant`.

Instances of **object types**, on the other hand, are typically more complex and maintain some sort of identity. Object types are rarely copied (cloned). If cloning is permitted, the operation is usually expensive and results in a new object (graph) that has a separate identity from the original.

The designers of `QObject` asserted an unequivocal “no copy” policy by designating its assignment operator and copy constructor `private`. This effectively prevents the compiler from generating assignment operators and copy constructors for `QObject`-derived classes. One consequence of this scheme is that any attempt to pass or return `QObject`-derived classes by value to or from functions results in a compile-time error.

EXERCISES: QOBJECT

1. Rewrite the `Contact` and `ContactList` from “Exercise: Contact List” in Chapter 4 so that they both derive from `QObject`.
When a `Contact` is to be added to a `ContactList`, make the `Contact` the child of the `ContactList`.
2. Port the client code you wrote for “Exercise: Contact List” to use the new versions of `Contact` and `ContactList`.

9.6 tr() and Internationalization

If you are writing a program that will ever be translated into another language (**internationalization**), Qt Linguist and Qt translation tools have already solved the problem of how to organize and where to put the translated strings. To prepare our code for translation, we use `QObject::tr()` to surround any translatable string in our application. The `tr()` function is generated for every `QObject` and places its international strings in its own “namespace,” which has the same name as the class.

`tr()` serves two purposes:

1. It makes it possible for Qt’s `lupdate` tool to extract all of the translatable string literals.
2. If a translation is available, and the language has been selected, the strings will actually be translated into the selected language at runtime.

If no translation is available, `tr()` returns the original string.



It is important that each translatable string is indeed fully inside the `tr()` function and extractable at compile time. For strings with parameters, use the `QString::arg()` function to place parameters inside translated strings. For example,

```
statusBar()->message  
(tr("%1 of %2 complete. progress: %3%%"  
.arg(processed).arg(total).arg(percent)));
```

This way, translations can place the parameters in different order in situations where language changes the order of words/ideas.

For a much more complete guide to internationalization, we recommend [Blanchette06], written by one of Linguist’s lead developers.

POINT OF DEPARTURE

There are other open-source implementations of signals and slots, similar to the Qt `QObject` model. One is called **xobject** (available at <http://sourceforge.net/projects/xobject>). In contrast to Qt, it does not require any `moc`-style preprocessing, but instead relies heavily on templates, so it is only supported by modern (post-2002) C++ compilers. The **boost** library (available from <http://www.boost.org>) also contains an implementation of signals and slots.

REVIEW QUESTIONS

1. What does it mean when object A is the parent of object B?
2. What happens to a `QObject` when it is reparented?
3. Why is the copy constructor of `QObject` not public?
4. What is the composite pattern?
5. How can `QObject` be both composite and component?
6. How can you access the children of a `QObject`?
7. What is an event loop? How is it initiated?
8. What is a signal? How do you call one?
9. What is a slot? How do you call one?
10. How are signals and slots connected?
11. How can information be transmitted from one object to another?
12. Deriving a class from `QObject` more than once can cause problems. How might that happen accidentally?
13. What is the difference between value types and object types? Give examples.

10

CHAPTER 10

Generics and Containers

This chapter covers more deeply the subject of generics. Generics are classes and functions that can operate just as easily on objects as primitive types. Qt container classes are generic, template-based classes, and we will see examples using lists, sets, and maps. Operators, managed containers, and resource sharing are also discussed.

10.1	Generics and Templates	214
10.2	Containers	219
10.3	Managed Containers, Composites, and Aggregates	221
10.4	Implicitly Shared Classes	224
10.5	Generics, Algorithms, and Operators. .	225
10.6	Serializer Pattern	227
10.7	Sorted Map Example	229

10.1 Generics and Templates

C++ supports four distinct categories of types:

1. Primitives: `int`, `char`, `float`, `double`, etc.
2. Pointers
3. Instances of `class/struct`
4. Arrays

Because there is no common base type for these four distinct type categories, writing generic functions and classes that can operate on multiple type categories would be very difficult without the use of **templates**. Templates provide a means for C++ to generate different versions of classes and functions with parameterized types and common behavior. They are distinguished by the use of the keyword `template`, and a template parameter enclosed in angle brackets `<>`.

A **template parameter** differs from a function parameter in that it can be used to pass not only variables and values, but also type expressions.

```
template <class T > class String { ... };
template <class T, int max > Buffer { ...
    T v[max];
};
String <char> s1;
Buffer <int, 10> intBuf10;
```

10.1.1 Function Templates

Function templates are used to create type-checked functions which all work on the same pattern. Example 10.1 defines a template function that raises a value of type T to the power exp by repeatedly applying the operator `*=`.

EXAMPLE 10.1 `src/templates/template-demo.cpp`

```
[ . . . . ]

template <class T> T power (T a, int exp) {
    T ans = a;
    while (--exp > 0) {
```

```

    ans *= a;
}
return (ans);
}

```

When the function is called, as shown in Example 10.2, different function bodies will be automatically generated by the compiler based on the argument types supplied in the function call. Even though the word `class` is in the template parameter, we can supply a class or a primitive type for T . The only limitation on the type T is that it must be a type for which the `operator* =` is defined.

EXAMPLE 10.2 `src/templates/template-demo.cpp`

```

[ . . . . ]

int main() {
    Complex z(3,4), z1;
    Fraction f(5,6), f1;
    int n(19);
    z1 = power(z,3);           ❶
    f1 = power(f,4);         ❷
    z1 = power<Complex>(n, 4); ❸
    z1 = power(n,5);         ❹
}

```

- ❶ First instantiation T is `Complex`.
- ❷ Second instantiation T is `Fraction`.
- ❸ Supply an explicit template parameter if the actual argument is not “specific” enough. This results in a call to a function that was already instantiated.
- ❹ Which version gets called?

Each time the compiler sees a template function used for the first time with a specific combination of parameter types, we say the template is **instantiated**. Subsequent uses of `power(Complex, int)` or `power(Fraction, int)` will be translated into ordinary function calls.

EXERCISES: FUNCTION TEMPLATES

1. Complete Example 10.2. In particular, write a generic `Complex` and `Fraction` class, and fix `main()` so that it works and uses those classes.
2. Write a template version of `swap()`, based on Example 5.13. Write client code to test it thoroughly.

3. Are there any types for which `swap()` does not work?
4. Specify the restrictions on the class parameter in your template `swap` function.

10.1.2 Class Templates

Like functions, classes can also use parameterized types. Class templates are used to generate generic containers of data. The parameter is the answer to the question, “Container of what?” All Qt container classes and, of course, all classes in the Standard Template Library (STL) are parameterized.

We will discuss a homemade example of a template stack class. Figure 10.1 shows a UML diagram of two template classes.

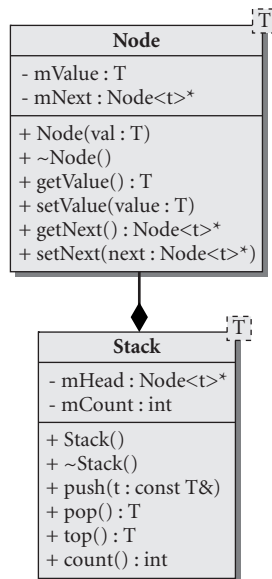


FIGURE 10.1 Template-based stack

UML locates the template parameter in a small offset box in the upper-right corner of the class box. Example 10.3 contains definitions for these classes.

EXAMPLE 10.3 `src/collections/stack/stack.h`

```

[ . . . . ]

#include <qstd.h>
template<class T> class Node {
public:
    Node(T invalue): m_Value(invalue), m_Next(0) {}
    ~Node() ;

```

```

    T getValue() const {return m_Value;}
    void setValue(T value) {m_Value = value;}
    Node<T>* getNext() const {return m_Next;}
    void setNext(Node<T>* next) {m_Next = next;}
private:
    T m_Value;
    Node<T>* m_Next;
};

template<class T> Node<T>::~~Node() {
    using namespace qstd;
    cout << m_Value << " deleted " << endl;
    if(m_Next) {
        delete m_Next;
    }
}

template<class T> class Stack {
public:
    Stack(): m_Head(0), m_Count(0) {}
    ~Stack<T>() {delete m_Head;}
    void push(const T& t);
    T pop();
    T top() const;
    int count() const;
private:
    Node<T> *m_Head;
    int m_Count;
};

```

Notice that template definitions, shown in Example 10.4 (classes *and* functions), appear in the header file. This is necessary for the compiler to generate code from a template declaration. Also notice the required template declaration code, `template<class T>`. This code must precede each class or function definition that has a template parameter in its name.

EXAMPLE 10.4 `src/collections/stack/stack.h`

```

[ . . . . ]

template <class T> void Stack<T>::push(const T& value) {
    Node<T> *newNode = new Node<T>(value);
    newNode->setNext(m_Head);
    m_Head = newNode;
    ++m_Count;
}

template <class T> T Stack<T>::pop() {
    Node<T> *popped = m_Head;
    if (m_Head != 0) {

```

continued

```

        m_Head = m_Head->getNext();
        T retval = popped->getValue();
        popped->setNext(0);
        delete popped;
        --m_Count;
        return retval;
    }
    return 0;
}

```

The creation of objects is handled generically in the template function, `push()`. The destructor for the `Node<T>` class recursively deletes `Node` pointers until it reaches one with a zero `m_Next` pointer.¹ Controlling creation and destruction of `Node<T>` objects this way enables `Stack<T>` to completely manage its dynamic memory. Example 10.5 contains some client code to demonstrate these classes.

EXAMPLE 10.5 `src/collections/stack/main.cpp`

```

#include <QDebug>
#include <QString>
#include "stack.h"

int main() {
    Stack<int> intstack1, intstack2;
    int val;
    for(val = 0; val < 4; ++val) {
        intstack1.push(val);
        intstack2.push(2 * val);
    }
    while (intstack1.count()) {
        val = intstack1.pop();
        qDebug() << val;
    }
    Stack<QString> stringstack;
    stringstack.push("First on");
    stringstack.push("second on");
    stringstack.push("first off");
    QString val2;
    while (stringstack.count()) {
        val2 = stringstack.pop();
        qDebug() << val2;
    }
    qDebug() << "Now intstack2 will self destruct.";
    return 0;
}

```

¹ This is a consequence of the fact that calling `delete` on a pointer automatically invokes the destructor associated with that pointer.

Output :

```

3 deleted
3
2 deleted
2
1 deleted
1
0 deleted
0
first off deleted
"first off"
second on deleted
"second on"
First on deleted
"First on"
Now intstack2 will self destruct.
6 deleted
4 deleted
2 deleted
0 deleted

```

EXERCISES: CLASS TEMPLATES

1. Place the function definitions for `Stack` in a separate file (`stack.cpp`), modify the project file appropriately, and then build the app. Explain the results.
2. How general is this application (i.e., what conditions must the class `T` satisfy in order to be used here)?
3. What limits the size of a `Stack<T>`?
4. Write a template `Queue<T>` class and client code to test it.

10.2 Containers

Qt's container classes are used to collect value types (things that can be copied), including pointers to object types (but not object types themselves). Qt containers are defined as template classes which leave the collected type unspecified. Each data structure is optimized for different kinds of operations. In Qt 4, there are several template container classes to choose from.

- `QList<T>` is implemented using an array, with space preallocated at both ends. It is optimized for index-based random access and, for lists with less than a thousand items, it also gives good performance with operations like `prepend()` and `append()`.

- `QStringList` is a convenience class that is derived from `QList<QString>`.
- `QLinkedList<T>` is optimized for sequential access with iterators and quick, constant-time inserts anywhere in the list. Sorting and searching are slow. It has several convenience functions for frequently used operations.
- `QVector<T>` stores its data in contiguous memory locations and is optimized for random access by index. Generally, `QVector` objects are constructed with an initial size. There is no automatic preallocation of memory at either end, so insertions, appends, and prepends are expensive.
- `QStack<T>` is publicly derived from `QVector<T>`, so the public interface of `QVector` is available to `QStack` objects. However, the last-in-first-out semantics are implemented by the `push()`, `pop()`, and `top()` functions.
- `QMap<Key, T>` is an ordered **associative container** that stores (key, value) pairs and is designed for fast lookup of the value associated with a key and for easy insertions. It keeps its keys in sorted order, for fast searching and subranging, by means of a skip-list dictionary² that is probabilistically balanced and uses memory very efficiently. The `Key` type must have an `operator<()` and `operator==(())`.
- `QHash<Key, T>` is an associative container that uses a hash table to facilitate key lookups. It provides very fast lookups (exact key match) and insertions, but slow searching and no sorting. The `Key` type must have an `operator==(())`.
- `QMultiMap<Key, T>` is a subclass of `QMap`, and `QMultiHash<Key, T>` is a subclass of `QHash`. These two classes allow multiple values to be associated with a single key.
- `QCache<Key, T>` is also an associative container but it provides fastest access to recently used items and automatic removal of infrequently used items based on cost functions.
- `QSet<T>` stores values of type `T` using a `QHash` with keys in `T` and a dummy value associated with each key. This arrangement optimizes lookups and insertions. `QSet` has functions for the usual set operations (e.g., union, intersection, set difference, etc.). The default constructor creates an empty set.

A type parameter `T` for a template container class or key type for an associative container must be an **assignable data type** (i.e., a value type; see Section 9.5). This

² <ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf>

means that it must have a public default constructor, copy constructor, and assignment operator.

Basic types (e.g., `int`, `double`, `char`, etc.) and pointers are assignable. Some Qt types are assignable (e.g., `QString`, `QDate`, `QTimer`). `QObject` and types derived from `QObject` are *not assignable*. If you need to group objects of some non-assignable type, you can define a container of pointers, e.g., `QList<QFile*>`.

10.3 Managed Containers, Composites, and Aggregates

Value containers are containers of uniform (same-typed) values, for example, `QString`, `byte`, `int`, `float`, etc. Pointer containers are containers of pointers to (polymorphic commonly typed) objects. They can be **managed** or **unmanaged**.

Both kinds of containers can grow at runtime by allocating additional heap memory as needed. This is always done in an exception-safe way, so you don't need to worry about possible memory leaks.

In the case of pointer containers to heap objects, however, one must decide which class is responsible for managing the heap objects. UML diagrams can distinguish between managed and unmanaged containers by using **composite** (filled diamond) and **aggregate** (empty diamond) connectors, as shown in Figure 10.2.

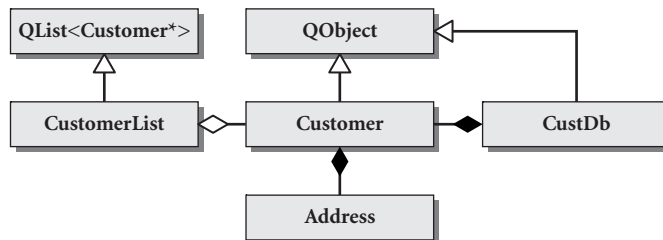


FIGURE 10.2 Aggregates and compositions

In general, we can say that a managed container is a composite, because the container manages its pointed-to objects. In other words, when a composite is destroyed, it destroys (cleans up) its entire self (because the smaller objects are part of its composition).

When one object embeds another as a sub-object, it is also considered a composition.

In Figure 10.2, there are two kinds of `Customer` containers, `CustomerList` and `CustDb`. `CustDb` and `CustomerList` both reuse template containers.

`CustomerList` objects are aggregates—temporary structures to hold the results of a query, or a user selection. `CustDb`, on the other hand, is a singleton composite that manages all of the `Customer` objects that exist.

In the case of the `Customer` and `Address` relationship, this diagram indicates that one or more `Address` objects should be associated with a particular `Customer`. When the `Customer` object is destroyed, it is reasonable to destroy all of its `Address` objects at the same time. Thus, the `Customer` object manages its `Addresses`, which gives us another example of a composite relationship.



This suggested design does impose some limitations on possible use of `Address`; in particular, there is no easy way to find all customers at a particular address. If `Address` and `Customer` were independently managed, then we could form bidirectional relationships between the classes.

Typically, a **managed container** deletes any heap objects it “owns” when the container itself is destroyed. With a Qt container of pointers, one can use `qDeleteAll(container)`, an algorithm that calls `delete` on each element in the container.

Copying a managed container can be defined in a number of ways:

- For some containers, the feature might be disabled.
- For others, it might be called a deep copy, where all contained objects are cloned and placed in the new container.
- Another approach, taken with the design of Qt containers, is implicit sharing, explained in the next section.

When a container only provides an indexing or reference navigation mechanism to its objects we call it an **aggregate container**.

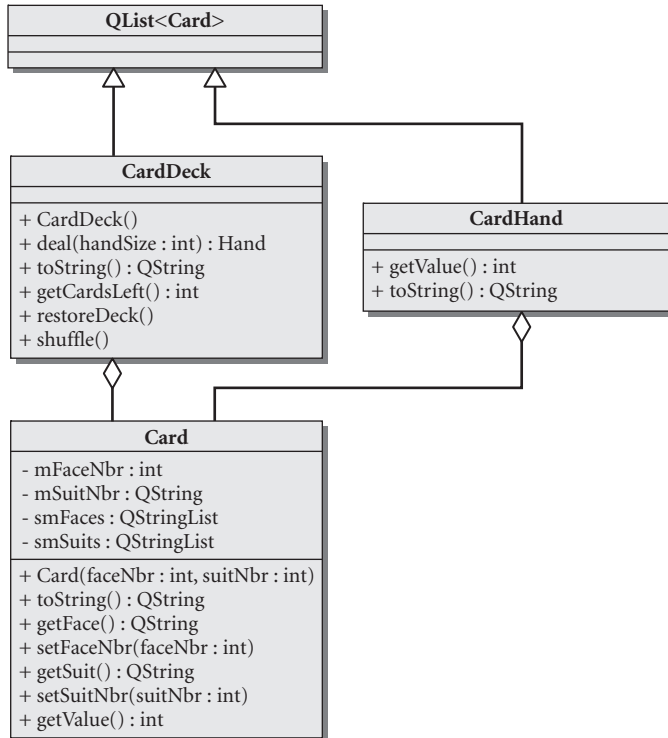
In this case, the container does not manage its objects, it only provides a convenient way to access them. When an aggregate container is copied, only references to the collected objects are copied. When an aggregate container is deleted, only the references are removed. There is no impact on the underlying objects in the container.



A managed container is a composition, and an unmanaged container of objects is usually (but not always) represented in a UML diagram as aggregation.

EXERCISE: MANAGED CONTAINERS, COMPOSITIONS AND AGGREGATES

This exercise involves designing some data types to represent a deck and a hand of cards. The following UML diagram suggests one way of representing them.



Here are some hints:

- The `CardDeck` constructor generates a complete deck of 52 cards in a convenient order.
- `CardDeck::deal(int k)` should use the `random()` function from `<stdlib.h>` to pick `k` cards from the deck (removing each one from the deck after it is picked) to fill a `CardHand` object.
- Initialize the `random()` function from the system clock so that the results will be different each time you run the app. The syntax is

```
srandom(time(0));
```

- Evaluate the hand using the rules of the game of bridge: Ace = 4, King = 3, Queen = 2, Jack = 1; all other cards have zero value. Use this formula to calculate the return values for the `getValue()` functions.
- Example 10.6 gives a piece of client code that you can start with for testing.

EXAMPLE 10.6 `src/cardgame/datastructure/cardgame-client.cpp`

```
#include "carddeck.h"
#include <qstd.h>
using namespace qstd; ❶

int main() {
    CardDeck deck;
    CardHand hand;
    int handSize, playerScore, progScore;
    cout << "How many cards in a hand? " << flush;
    handSize = promptInt();
    do {
        hand = deck.deal(handSize);
        cout << "Here is your hand:" << endl;
        cout << hand.toString() << endl;
        playerScore = hand.getValue();
        cout << QString("Your score is: %1 points.")
             .arg(playerScore) << endl;
        // Now a hand for the dealer:
        hand = deck.deal(handSize);
        progScore = hand.getValue();
        cout << "Here is my hand:" << endl;
        cout << hand.toString() << endl;
        cout << QString("My score is: %1 points.")
             .arg(progScore) << endl;
        cout << QString("%1 win!!")
             .arg((playerScore > progScore)?"You":"I") << endl;
    } while (more("hand"));
}
```

❶ for `cout`, `endl`, and `more()`

10.4 Implicitly Shared Classes

A hybrid kind of managed container is the **implicitly shared** container. Container classes in the Qt library implement “lazy copy on write,” or **implicit sharing**. This means they have reasonably fast³ copy constructors and assignment operators. Only when the copy is actually modified are the collected objects cloned from the original container. That is when there will be a time/memory penalty.

³ Operations should be “not much slower” than the time it takes to do one pointer-copy and one integer increment.

`QString` and `QStringList` are both implemented this way, meaning that it is fast to pass and return these objects by value. If you need to change values stored in the container from inside a function, you can pass the container by reference. It is still faster to pass by `const` reference, which allows C++ to optimize out the copy operation entirely. With `const` reference, the function cannot make changes to the container at all.

Implicitly shared classes work by reference-counting, to prevent the accidental deletion of shared managed objects. Since implicitly shared memory is encapsulated, the user of the class does not need to be concerned with reference counts or direct memory pointers.

We look in more detail at the implementation of a reference counted class in Section 24.2.

10.5 Generics, Algorithms, and Operators

Overloading operator symbols makes it possible to define a common interface for our classes that is consistent with that of the basic types. Many generic algorithms take advantage of this by using the common operators to perform basic functions such as comparison.

The `qSort()` function is a generic algorithm that is implemented using the heap sort algorithm. In Example 10.7, we show how it can be used on two similar but different containers.

`qSort()` can be applied to any Qt container of objects that have publicly defined functions `operator<()` and `operator==(())`. Containers of built-in numeric types can also be sorted with this function.

EXAMPLE 10.7 `src/collections/sortlist/sortlist4.cpp`

```
#include <QList>
#include <assertequals.h>
#include <QtAlgorithms> // for qSort()
#include <qstd.h>       // for cin and cout
using namespace qstd;

class CaseIgnoreString : public QString {
public:
    CaseIgnoreString(const QString& other = QString()) :
        QString(other) {}
```

continued

```

    bool operator<(const QString & other) const {
        return toLower() < other.toLower();
    }
    bool operator==(const QString& other) const {
        return toLower() == other.toLower();
    }
};

int main() {
    CaseIgnoreString s1("Apple"), s2("bear"),
                    s3 ("CaT"), s4("dog"), s5 ("Dog");

    ASSERT_TRUE(s4 == s5);
    ASSERT_TRUE(s2 < s3);
    ASSERT_TRUE(s3 < s4);

    QList<CaseIgnoreString> namelist;

    namelist << s5 << s1 << s3 << s4 << s2; ❶

    qSort(namelist.begin(), namelist.end());
    int i=0;
    foreach (QString stritr, namelist) {
        cout << QString("namelist[%1] = %2")
             << .arg(i++).arg(stritr) << endl;
    }

    QStringList strlist;
    strlist << s5 << s1 << s3 << s4 << s2; ❷

    qSort(strlist.begin(), strlist.end());
    cout << "StringList sorted: " + strlist.join(", ") << endl;
    return 0;
}

```

❶ Insert all items in an order that is definitely not sorted.

❷ The value collection holds QString, but we are adding CaseIgnoreString. A conversion is required.

`operator<<()`, which is the *left shift* operator from C, has been overloaded in the `QList` class to append items to the list.

Example 10.8 shows the output of this program.

EXAMPLE 10.8 `src/collections/sortlist/sortlist-output.txt`

```

namelist[0] = Apple
namelist[1] = bear
namelist[2] = CaT
namelist[3] = dog
namelist[4] = Dog
StringList sorted: Apple, CaT, Dog, bear, dog

```

Notice that the sorting order is case sensitive when we add `CaseInsensitiveString` objects to a `QStringList`. This is because a `CaseInsensitiveString` must be converted into a `QString` as it is added to `strlist`. Therefore, when `strlist`'s elements are compared, they are compared as `QString`s.

EXERCISES: GENERICS, ALGORITHMS, AND OPERATORS

1. A `QStringList` is a value container of objects that have lazy copy-on-write. In a way, it is like a pointer-collection, but smarter. In Example 10.7, a `CaseInsensitiveString` was added to a `QStringList`, which required a conversion. Does this require a copy of the actual string data? Why or why not?
2. Add some more functions to `ContactList`:

ContactList
+ operator +=(c : Contact)
+ operator -=(c : Contact)
+ sortByCategory()
+ sortByZip()

Operators `+=` and `-=` should add `()` and remove `()` respectively.

Write some client code that tests these functions.

10.6 Serializer Pattern

A **serializer** is an object that is responsible only for reading or writing objects. With `QDataStream`, it is already possible to serialize and deserialize all `QVariant`-supported types, including `QList`, `QMap`, `QVector`, and others. For other file formats, or more complex object models, we isolate the reading/writing in separate reader and writer classes. These classes are examples of the **Serializer pattern** [Martin 98].

A serializer that reads and writes to files should handle all of the file and/or stream initialization and cleanup. A serializer could also be used to send objects over a network from one socket to another.⁴ In that case it would be responsible for connecting and disconnecting to/from the appropriate socket.

⁴ Sockets are addressable entities, used as endpoints for sending and receiving data between computers. Qt has an abstract base class named `QAbstractSocket`, which provides the interface for working with various kinds of sockets, and a concrete `QTcpSocket` class, which provides a TCP (Transmission Control Protocol) socket.

In Figure 10.3, we have a UML diagram for two serializer classes. Together, they can read and write `ContactList` objects.

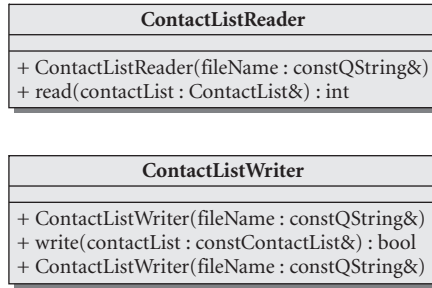


FIGURE 10.3 The Serializer pattern

In C++, serializer classes can have overloaded input/output operators, similar to `iostream` or `QTextStream`, so that we can use them with a familiar interface. To read a file into the container, it should be as easy as this:

```
ContactList cl;
ContactListReader reader("somefile.txt");
reader >> cl;
```

And analogously, when it is time to output, the client code should look like this:

```
ContactListWriter writer("somefile.txt");
writer << cl;
```

We achieve this by defining the following insertion operators:

```
ContactListReader& operator>>(ContactListReader& reader,
    ContactList& cl);
ContactListWriter& operator<< (ContactListWriter& writer,
    const ContactList& cl);
```

Example 10.9 shows how to implement customized i/o operators.

EXAMPLE 10.9 `src/containers/contact/serializer.cpp`

```
[ . . . . ]

ContactListReader& operator>>(ContactListReader& reader,
    ContactList& cl) {
    reader.read(cl);
    return reader;
}

ContactListWriter& operator<< (ContactListWriter& writer,
    const ContactList& cl) {
    writer.write(cl);
    return writer;
}
```


In Example 10.9, we defined global (non-member) operators. It would also be possible to write member function operators. The reason it is necessary to define non-member operators for `iostream` and `QTextStream` is because we cannot modify those classes to add member functions.

10.6.1 Benefits of the Serializer

By using the serializer, it will become much easier to change the file format, or the transport layer, with only small changes in client code. In addition, removing serialization code from the model makes the model simpler and easier to maintain.

EXERCISES: SERIALIZER PATTERN

1. Revisiting the Contact List exercise from Section 4.3, complete the implementation of `ContactListWriter` and `ContactListReader` classes for the `ContactList`, following the Serializer pattern, so that you can read and write a `ContactList` to a file.

When each `Contact` is serialized, it should be written on a single line, in a tab-separated format (tabs separating each field in the record). Therefore, a serialized `ContactList` should have a sequence of lines, each line representing a `Contact`.

2. Write a main program that supports, at minimum, the following command line arguments. Feel free to add others after these are working.

```
Contact List Test Driver usage
contact [-i inputfile] [-c] [-o outputfile] [-p]
  -i read contact list from specified inputfile into ContactList
  -c generate 10 more random contacts and add to ContactList
  -o write ContactList list to specified output file
  -p print ContactList to standard output
```

If an invalid option is supplied, or if no options are supplied, it should print the above "Contact List Usage".

10.7 Sorted Map Example

As we mentioned earlier, `QMap` is an associative array that maintains key sorting-order as items are added and removed. Key-based insertions and deletions are fast ($\log(n)$) and iteration is done in key order.

`QMap` is a value container, but pointers are simple values, so we can use a `QMap` to store pointers to heap allocated `QObject`s. By default, value containers do not manage heap objects, so to avoid memory leaks we must ensure they are destroyed

at the proper time. Figure 10.4 describes a class that extends a `QMap` to contain information about textbooks. By deriving from `QMap`, the entire public interface of `QMap` becomes part of the public interface of `TextbookMap`. We only added a destructor plus two convenience functions to facilitate adding and displaying Textbooks in the container.

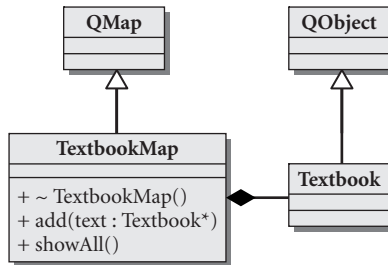


FIGURE 10.4 TextBookMap

In `TextbookMap`, defined in Example 10.10, we store ISBN numbers as keys, and pointers to `Textbook` objects as values. Example 10.10 shows the class definitions.

EXAMPLE 10.10 `src/containers/qmap/textbook.h`

```

#ifndef _TEXTBOOK_H_
#define _TEXTBOOK_H_

#include <QObject>
#include <QString>

class Textbook : public QObject {
    Q_OBJECT
public:
    Textbook(QString title, QString author, QString
isbn, uint year);
    [ . . . . ]
private:
    uint m_YearPub;
    QString m_Title, m_Author, m_Isbn;
};

class TextbookMap : public QMap<QString, Textbook*> {
public:
    ~TextbookMap();
    void add(Textbook* text);
    void showAll() const;
};
#endif

```

EXAMPLE 10.11 `src/containers/qmap/qmap-example.cpp`

```
[ . . . . ]

TextbookMap::~TextbookMap() {
    cout << "Destroying TextbookMap ..." << endl;
    foreach (QString key, keys()) ❶
        delete value(key);      ❷
    clear();
}

void TextbookMap::add(Textbook* text) {
    insert(text->getIsbn(), text);
}

void TextbookMap::showAll() const {
    foreach (QString key, keys()) {
        Textbook* tb = value(key);
        cout << '[' << key << ']' << ":"
             << tb->toString() << endl;
    }
}
```

- ❶ keys() is a QMap function.
- ❷ Get and delete a pointer from the map

In Example 10.11, the destructor iterates through the `QMap`, deleting each pointer. This is necessary for a value container to manage its objects.

Notice in the client code, shown in Example 10.12, that when we `remove()` a pointer from the `TextbookMap` we also remove its responsibility for deleting that pointer.

EXAMPLE 10.12 `src/containers/qmap/qmap-example.cpp`

```
[ . . . . ]

int main() {
    Textbook* t1 = new Textbook("The C++ Programming Language",
                               "Stroustrup", "0201700735", 1997);
    Textbook* t2 = new Textbook("XML in a Nutshell", "Harold",
                               "0596002920", 2002);
    Textbook* t3 = new Textbook("UML Distilled", "Fowler",
                               "0321193687", 2004);
    Textbook* t4 = new Textbook("Design Patterns", "Gamma",
                               "0201633612", 1995);
    {
        ❶
```

continued

```
        TextbookMap m;  
        m.add(t1);  
        m.add(t2);  
        m.add(t3);  
        m.add(t4);  
        m.showAll();  
        m.remove (t3->getIsbn()); ❷  
    }                               ❸  
    cout << "After m has been destroyed we still have: \n"  
        << t3->toString() << endl;  
    return 0;  
}
```

- ❶ inner block for demonstration purposes
 - ❷ removed but not deleted
 - ❸ end of block, local variables destroyed.
-

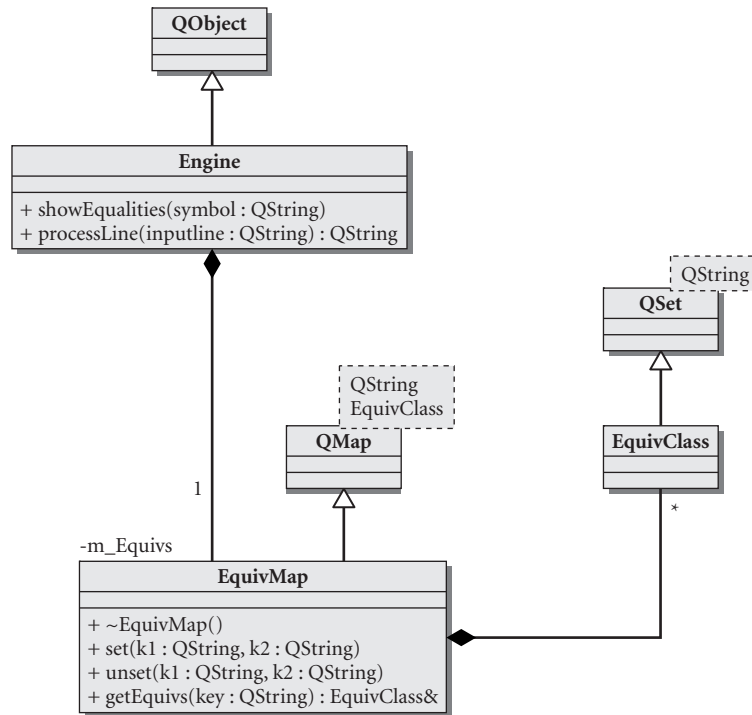
When `TextbookMap::ShowAll()` iterates through the container, we can see from the output in Example 10.13 that the `Textbooks` have been placed in order by ISBN (the key).

EXAMPLE 10.13 `src/containers/qmap/qmap-example-output.txt`

```
src/containers/qmap> ./qmap  
[0201633612]:Title: Design Patterns; Author: Gamma; ISBN:  
0201633612; Year: 1995  
[0201700735]:Title: The C++ Programming Language; Author:  
Stroustrup; ISBN: 0201700735; Year: 1997  
[0321193687]:Title: UML Distilled; Author: Fowler; ISBN:  
0321193687; Year: 2004  
[0596002920]:Title: XML in a Nutshell; Author: Harold; ISBN:  
0596002920; Year: 2002  
Destroying TextbookMap ...  
After m has been destroyed we still have:  
Title: UML Distilled; Author: Fowler; ISBN: 0321193687; Year: 2004  
src/containers/qmap>
```

EXERCISE: QSETS AND QMAPS

This exercise requires you to use a map of sets to implement a dynamic equivalence relation.



An **equivalence relation** is a boolean operator, *equiv*, on a set *S* of symbols that has three properties:

1. Reflexivity: For any symbol *s* in *S*, we always have

s equiv s

2. Symmetry: For any symbols *s* and *t* in *S*,

if s equiv t then t equiv s

3. Transitivity: For any symbols *s*, *t*, and *u* in *S*,

if s equiv t and t equiv u then s equiv u

Another example of an equivalence relation is the operator `==` relationship between pointers. This one is very easy to check, because their addresses also must be equal. If the set members are different symbols with different values, we need to keep track of the sets themselves in order to determine which symbols are related to each other.

An equivalence relation, *equiv*, partitions the set *S* of symbols into subsets called equivalence classes. An equivalence class is a set of symbols that are equivalent to one another. It is easy to see that two equivalence classes are either equal or disjoint.

1. Write a program that repeatedly asks the user to enter assertions or commands, and keeps track of equivalency sets for all QStrings that it sees. At any time, the user should be able to see the contents of any equivalence class. In other words, `processLine()` should expect strings of the following form:
 - a. To make an equivalence assertion between two strings: `string1=string2`
 - b. To list the contents of `string1`'s `EquivClass`: `string1 (no = symbol in the line)`
2. Add another function, `takeback(int n)`, which takes an integer that refers to the *n*th assertion. The function should “undo” the *n*th assertion that was performed, fixing up all `EquivClass` sets in the equivalence engine. After this, list the updated equivalences of both symbols involved.

Have the `processLine()` function scan for messages of the form “take-back *n*” and call this `takeback()` in response.

REVIEW QUESTIONS

1. Explain an important difference between a template parameter and a function parameter.
2. What does it mean to instantiate a template function? Describe one way to do this.
3. Normally, you need to place template definitions in header files. Why is this?
4. Qt's container classes are used to collect value types. What kinds of things are not appropriate to store by value in a value collection?
5. Which containers provide a mappings from key to value? List and describe at least two, and tell how they are different from each other.
6. What does it mean for a container to "manage" its heap objects? How does a container of pointers to heap objects become a "managed container"?

11

CHAPTER 11

Qt GUI Widgets

This chapter provides an overview of the GUI building blocks, called widgets, that are part of the Qt library and includes some simple examples of how to use them.

11.1	Widget Categories	239
11.2	QMainWindow and QSettings	240
11.3	Dialogs	244
11.4	Images and Resources	248
11.5	Layout of Widgets	251
11.6	QActions, QMenus, and QMenuBar	260
11.7	QActions, QToolbars, and QActionGroups	262
11.8	Regions and QDockWidgets	270
11.9	Views of a QStringList	272

Widgets, objects of classes derived from `QWidget`, are reusable building blocks with a visual representation on the screen. The common features of a `QWidget` are shown in Figure 11.1.

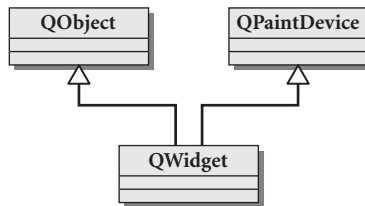


FIGURE 11.1 `QWidget`'s Heritage

`QWidget` is a class that uses multiple inheritance (see Section 23.3). A `QWidget` *is a* `QObject`, and thus can have parents, signals, slots, and managed children. A `QWidget` *is a* `QPaintDevice`, the base class of all objects that can be “painted.”

`QWidgets` interact with their children in interesting ways. A widget that has no parent is called a **window**. If one widget is a parent of another widget, the boundaries of the child widget lie completely within the boundaries of the parent. The contained child widget is displayed according to layout rules (see Section 11.5).

A `QWidget` can handle events by responding to signals from various entities in the window system (e.g., mouse, keyboard, other processes, etc.). It can paint its own rectangular image on the screen. It can remove itself from the screen in a way that respects whatever else is on the screen at the moment.

A typical desktop GUI application can contain many (hundreds is not unusual) different `QWidget`-derived objects, deployed according to parent-child relationships and arranged according to the specifications of the applicable layouts.

A `QWidget` is considered to be the simplest of all GUI classes, because it is rendered as an empty box. The class itself is quite complex, containing hundreds of functions. When you reuse `QWidget` and its subclasses you are standing on the shoulders of giants, because every `QWidget` is built on top of layers of Qt code,

which in turn are built on top of *different* layers of native widget libraries, depending on your platform (X11 in Linux, Cocoa on MacOS, and Win32 in Windows).

In fact, `QWidget` is a Façade for the widget classes from each of these native window libraries (see Section 16.3).

11.1 Widget Categories

Qt widgets can be categorized in a number of ways to make it easier to find classes you are likely to use. The more complex widgets may cross over into more than one category. This section provides a brief overview of some of the classes we are likely to use as we get started with GUI programming.

There are four categories of widgets. Button widgets, in “Windows style,” are shown in Figure 11.2.

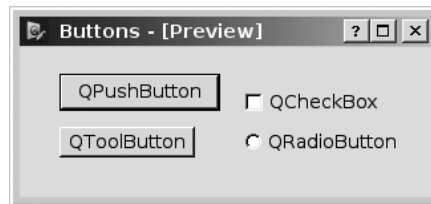


FIGURE 11.2 Button widgets, in “Windows style”

Input widgets, in “Plastique style,” are shown in Figure 11.3.

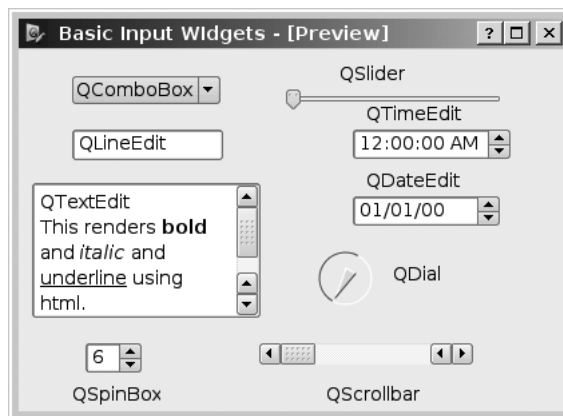


FIGURE 11.3 Input widgets, in “Plastique style”

Display widgets are non-interactive widgets, such as `QLabel`, `QProgressBar`, and `QPixmap`.

Container widgets, such as the `QMainWindow`, `QFrame`, `QToolBar`, `QTabWidget`, and `QStackedWidget`, contain other widgets.

These widgets are used as building blocks to create other more complex widgets, such as:

- Dialogs for asking the user questions or popping up information, such as the `QFileDialog`, `QInputDialog`, and `QErrorMessage`.
- Views that provide displays of collections of data such as `QListView`, `QTreeView`, and `QTableView`.

In addition, there are some Qt classes that do not have any graphical representation (so they are not widgets) but are used in GUI development. They include

- **Qt Data types:** `QPoint` and `QSize` are popular types to use when working with graphical objects.
- **Controller classes:** `QApplication` and `QAction` are both objects that manage the GUI application's control flow.
- **Layouts:** These are objects that dynamically manage the layout of widgets. There are specific layout varieties: `QHBoxLayout`, `QVBoxLayout`, `QGridLayout`, etc.
- **Models:** The `QAbstractItemModel` and its derived classes `QAbstractListModel` and `QAbstractTableModel` are part of Qt's model/view framework, and are used as base classes for classes that represent data for a `QListView`, `QTreeView`, or `QTableView`.
- **Database models:** These are for use with `QTableView` (or other customized view classes) using SQL Databases as data sources: `QSqlTableModel` and `QSqlRelationalModel`.

To see more widgets rendered in different styles, check out TrollTech's Qt Widget Gallery¹, which contains a variety of screenshots and source code for rendering the widgets in different styles.¹

11.2 QMainWindow and QSettings

Most `QApplications` manage a single `QMainWindow`. As Figure 11.4 shows, the `QMainWindow` has some features that are common to most desktop applications:

- A central widget
- Menu bar

¹ <http://oop.mcs.suffolk.edu/qtdocs/gallery.html>

- Status bar
- Dock regions

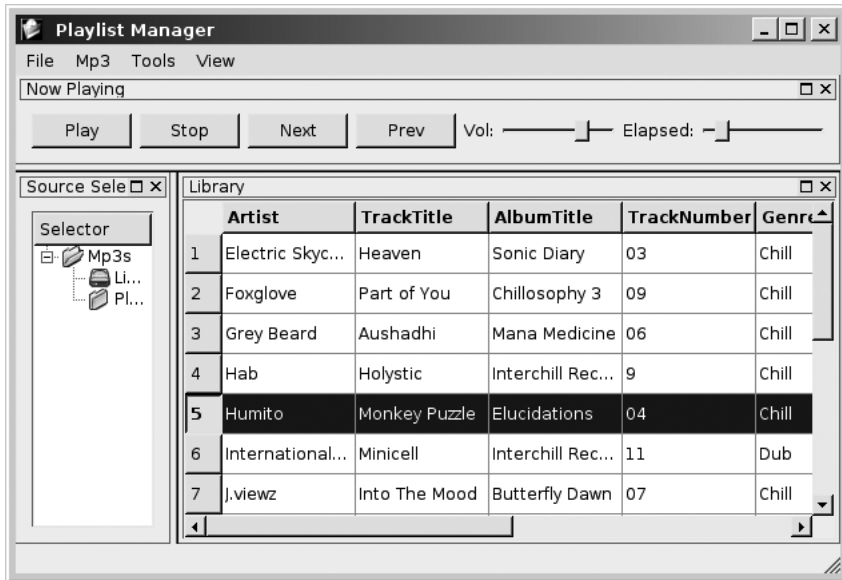


FIGURE 11.4 A main window

Because the `QMainWindow` is the parent of all other widgets (in that main window), it is common practice to extend that class for some applications, as shown in Example 11.1.

EXAMPLE 11.1 src/widgets/mainwindow/mymainwindow.h

```
[ . . . . ]
class MyMainWindow : public QMainWindow {
    Q_OBJECT

public:
    MyMainWindow();
    void closeEvent(QCloseEvent *event); ❶

protected slots:
    virtual void newFile();
    virtual void open();
    virtual bool save();
[ . . . . ]
```

❶ overridden from base class to capture when the user wants to close the window

11.2.1 QSettings: Saving and Restoring Application State

All modern desktop applications have a way for users to configure the settings. The settings/preferences/options need to be persistent. The mechanism for that is included with `QSettings`. As you develop a new `QMainWindow` application, the first persistent settings you may want to save will probably be the window size and position. You may also want to save the names of the most recent documents that were opened by the application.

`QSettings` is a persistent map of key/value pairs. It is a `QObject` and uses `QObject`'s property interface, `setValue()` and `value()`, to set and get its values. It can be used to store any data that needs to be remembered across multiple executions.

The `QSettings` constructor has two `QString` parameters: one for the organization name and one for the application name. You can establish defaults for these values with the two `QCoreApplication` functions, `setOrganizationName()` and `setApplicationName()`, after which you can use the default `QSettings` constructor. Each combination of names defines a unique persistent map that does not clash with settings from other-named Qt applications.

Monostate Pattern

A class that allows multiple instances to share the same state is an implementation of the **Monostate pattern**. Two instances of `QSettings` with the same organization/application name can be used to access the same persistent map data. This makes it easy for applications to access common settings from different source files.

`QSettings` is an implementation of the Monostate pattern.

The actual mechanism for the persistent storage of `QSettings` data is implementation dependent and quite flexible. Some possibilities for its storage include the Win32 registry (in Windows) and your `$HOME/.qt` directory (in Linux). For more detailed information, see the Qt `QSettings` API documentation.

`QMainWindow::saveState()` returns a `QByteArray` that contains information about the main window's toolbars and dockwidgets. To do this it makes use of the `objectName` property for each of those subwidgets, thus making it important that each name be unique. `saveState()` has an optional `int` `versionNumber`

parameter. The `QSettings` object stores that `QByteArray` with the key string "state".

`QMainWindow::restoreState()` takes a `QByteArray`, presumably created by `saveState()`, and uses the information that it holds to put the toolbars and dockwidgets into a prior arrangement. It too has an optional `versionNumber` parameter. We show the use of these functions in Example 11.2.

EXAMPLE 11.2 `src/widgets/mainwindow/mymainwindow.cpp`

```
[ . . . . ]

void MyMainWindow::readSettings() {
    QSettings settings("objectlearning.net", "Qt4 Sample Main"); ❶
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    QByteArray state = settings.value("state", QByteArray()).toByteArray();
    restoreState(state);
    resize(size);
    move(pos);
}

void MyMainWindow::writeSettings() {
    /* Save position/size of main window */

    QSettings settings("objectlearning.net", "Qt4 Sample Main");
    settings.setValue("pos", pos());
    settings.setValue("size", size());
    settings.setValue("state", saveState());
}

```

❶ The constructor takes the organization name and the app name as arguments.

By placing the “root program logic” in a derived class of `QMainWindow` or `QApplication`, the `main()` becomes quite simple, as shown in Example 11.3.

EXAMPLE 11.3 `src/widgets/mainwindow/mainwindow-main.cpp`

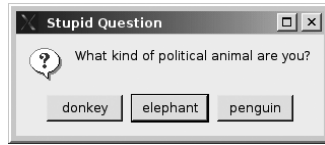
```
#include "mymainwindow.h"
#include <QApplication>

int main( int argc, char ** argv ) {
    QApplication a( argc, argv );
    MyMainWindow mw;
    mw.show();
    return a.exec();
}

```

11.3 Dialogs

A **dialog box** pops up, shows the user something, and gets a response back from the user. It's a brief interaction that can (optionally) force the user to respond.



`QDialog`, the base class for dialogs, has a `bool` attribute named **modal**. If `modal` is set to `true`, the dialog box blocks input to all other visible windows in its parent. A modal dialog box must be dismissed (by clicking on one of its buttons) before the application can proceed.

In general, dialog boxes should be used only for obtaining or communicating important information that should (or must) be dealt with before the program can proceed.

The `QMessageBox` (derived from `QDialog`) class produces a modal dialog with a brief message (or question), a distinctive icon, and from one to three buttons. Each of the icons characterizes a particular severity level and corresponds to a particular kind of message box.



`QMessageBox` has a number of static convenience functions that cause these kinds of dialogs to pop up. A short application demonstrating the use of the various kinds of `QMessageBox` is shown in Examples 11.4 and 11.5.

EXAMPLE 11.4 `src/widgets/dialogs/messagebox/dialogs.h`

```
#ifndef APPWINDEMO_H
#define APPWINDEMO_H

#include <QMainWindow>
class Dialogs : public QMainWindow {
    Q_OBJECT
public:
    Dialogs();
```



```

public slots:
    void askQuestion();
    void askDumbQuestion();
private:
    QString answer;

};
#endif

```

EXAMPLE 11.5 src/widgets/dialogs/messagebox/dialogs.cpp

```

[ . . . . ]

void Dialogs::askQuestion() {
    bool done=false;
    QString q1("Who was Canadian Prime Minister in 1847?"),
            a0("John A. Macdonald"), a1("Alexander Mackenzie"),
            a2("Sir Wilfred Laurier");
    while (!done) {
        int ans = QMessageBox::
            question( this, ❶
                    "Difficult Question",
                    q1, a0, a1, a2,
                    0, ❷
                    -1 ); ❸
        if (ans > 0) return; ❹
        switch( ans ) {
        case 0:
            answer = a0;
            break;
        case 1:
            answer = a1;
            break;
        case 2:
            answer = a2;
            break;
        }
        QString q2(QString("Your answer was: %1."
                          " That is incorrect.\n Try again?")
                  .arg(answer));
        int again = QMessageBox::
            question(this, "Your Score", q2,
                   "No", "Yes", "I give up - what's the answer?");

        if ( again < 1 ) {
            return;
        }
    }
}

```

continued

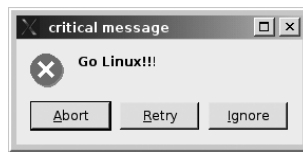
```

    }
    if (again == 2) {
        QMessageBox::
            information(this, "Ha Ha!",
                "There was no prime minister until 1867",
                "Grrrrr.....");
        return;
    }
}
}

```

- ❶ Message box is modal relative to its parent.
- ❷ default value
- ❸ Value sent on “esc” or “windowclose”.
- ❹ If window was closed, no QButton was pressed.

When you build and run this application, notice that once the critical dialog is on-screen, you cannot interact with the main window at all—the pull down Questions menu is not receiving events, and even the window manager can’t kill the main window. This is the power of a modal dialog.



11.3.1 Input Dialogs and Widgets

When a user enters text into a box, it is an input widget, probably a `QLineEdit` (for single lines), a `QComboBox` (for a choice of values), `QSpinBox` (for a number), or `QTextEdit` (for multiple lines of text). These widgets are never “solitary” because there needs to be at least a label nearby that tells the user what information is required.

Input dialogs are higher-level widgets that group together the lower-level input widgets with labels and decorations for convenience. `QInputDialog` has a label, an input widget (such as a `QLineEdit`), and, like the message dialogs, one to three buttons.

`QInputDialog` has four configurable static functions that determine the characteristics of the data entry widget. It can return a `QString`, an `int`, or a `double`. Example 11.6 is a short application that supplies the static function `getItem()` with a `QStringList` to store a set of possible responses and returns the `QString` selected by the user.

EXAMPLE 11.6 `src/widgets/dialogs/inputdialog/inputdialog.cpp`

```

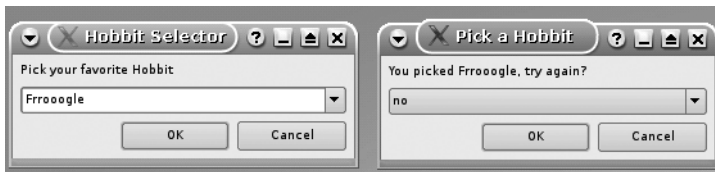
#include <QInputDialog>
#include <QStringList>
#include <QApplication>

int main(int argc, char** argv) {
    QApplication app(argc, argv);
    app.setQuitOnLastWindowClosed(false);
    QStringList hobbits, yesno;
    hobbits << "Frodo" << "Bilbo" << "Samwise" < "Merry" << "Pippin";
    yesno << "yes" << "no";
    QString outcome, more, title("Hobbit Selector");
    QString query("Pick your favorite Hobbit");
    do {
        QString pick =
            QInputDialog::getItem(0, title, query, hobbits);
        outcome = QString("You picked %1, try again?").arg(pick);
        more = QInputDialog::
            getItem(0, "Pick a Hobbit", outcome, yesno, 1, false);
    } while (more == "yes");
}

```

The first call to `getItem()` uses the overloaded version that accepts a `QStringList`. This puts a `QComboBox` (editable by default) on the screen, pre-filled with the values from the `QStringList`. The user can then select one of the choices or type in a new string.

We supplied six arguments in the second call to `getItem()`, resulting in a non-editable combobox. Here are two screenshots of the running program.



By default, Qt expects a GUI app to have a single top-level window (i.e., a window with no parent). Our Hobbit selection app has two. This can cause some confusion about when the program is finished.

We inserted the following function call to address this problem:

```
app.setQuitOnLastWindowClosed(false);
```

EXERCISE: DIALOGS

Make the following modifications of the Hobbit Selection app.

1. Remove the line


```
app.setQuitOnLastWindowClosed(false);
```

 and see if you can explain the results.
2. Change the app so that each time the user, by editing one of the selections in the list, creates a new Hobbit name, that name becomes one of the (sorted) choices for the next iteration.
3. In both of the `getItem()` dialogs of the Hobbit Selection app, there is no difference between OK and Cancel. What should that difference be? As Captain Picard would say, "Make it so."

11.4 Images and Resources

Graphic images can be used to add visual impact to various applications. In this section, we show how to build an application or a library that includes graphic images in the project.

Qt 4 enables projects to make use of binary resources, such as images, sounds, icons, text in some exotic font, and so forth. Resources such as these are generally stored on disk in separate binary files. The advantage of incorporating binary dependencies in the actual project is that they can then be addressed using paths that do not depend on the local file system, and they can “move” with the executable.

In the next examples, we will demonstrate how to create and reuse a library that includes images, one for each card in a deck of playing cards.

The first step is to list the binary files we wish to use in a **Resource Collection File**, an XML file that has the extension `.qrc`. The resource file in `libcards2` is automatically generated by a python script. Here is an abbreviated listing of it.

```
<!DOCTYPE RCC>
<RCC version="1.0">
<qresource>
<file alias="images/qh.png">images/qh.png</file>
<file alias="images/qd.png">images/qd.png</file>
<file alias="images/jc.png">images/jc.png</file>
[ ... ]
<file alias="images/jd.png">images/jd.png</file>
<file alias="images/ac.png">images/ac.png</file>
</qresource>
</RCC>
```

RCC is a file format, required by `qmake` and `rcc` (Qt's resource compiler), that generally consists of a list of `<file>` elements enclosed by `<qresource>` tags. Each `<file>` element contains a relative path and an optional file alias.

We must add a `RESOURCES` line in the project file that contains the name of the `qresource` file, as shown in Example 11.7.

EXAMPLE 11.7 `src/libs/cards2/cards2.pro`

```
# project file for libcards2

include ($$(CPPLIBS)/utils/common.pri)

TEMPLATE = lib
INCLUDEPATH += .
QT += gui
# For locating the files.
RESOURCES = cards2.qrc

SOURCES += cardpics.cpp
HEADERS += cardpics.h
DESTDIR=$$(CPPLIBS)
target.path=$$(CPPLIBS)
unix {
    UI_DIR = .ui
    MOC_DIR = .moc
    OBJECTS_DIR = .obj
}
```

When this project is built, the resource compiler generates an extra file named `cards2_qrc.cpp` that contains byte arrays defined in C++. This file gets compiled and linked into the project binary (executable or library) instead of the original files. The `DESTDIR` line specifies that the shared object files for `libcards2` will be located in `$$CPPLIBS` with the other libraries that we have built.

In `libcards2`, we defined a class named `CardPics`, shown in Example 11.8, which utilizes the `QPixmap` class. There are three Qt classes that facilitate the handling of images.

1. `QPixmap`: designed and optimized for drawing on the screen
2. `QImage`: designed for input and output operations and for direct pixel access
3. `QPImage`: designed to enable scalability

EXAMPLE 11.8 src/libs/cards2/cardpics.h

```
[ . . . . ]
class CardPics {
public:
    CardPics();
    static CardPics* instance();
    QPixmap get(QString card);
    static const QString values, suits;
protected:
    static QString fileName(QString card);
private:
    QMap<QString, QPixmap> m_pixmap;
};
[ . . . . ]
```

Attaching required binary data files to projects as resources makes the project more robust. The source code does not need to use nonportable pathnames for the resource files. To refer to a file that is stored as a resource, we can use the alias that was established in the RCC file, and precede it with the prefix, “:”. Each resource then appears (to Qt) to be located in a private virtual file system, rooted at :/. Example 11.9 shows some `QPixmap`s created with pathnames of this format.

EXAMPLE 11.9 src/libs/cards2/cardpics.cpp

```
[ . . . . ]
const QString CardPics::values="23456789tjqka";
const QString CardPics::suits="cdhs";

CardPics* CardPics::instance() {
    static CardPics* inst = 0;
    if (inst == 0)
        inst = new CardPics;
    return inst;
}

CardPics::CardPics() {
    foreach (QChar suit, suits) {
        foreach (QChar value, values) {
            QString card = QString("%1%2").arg(value).arg(suit);
            QPixmap pixmap(fileName(card));
            m_pixmap[card]= pixmap;
        }
    }
}

QString CardPics::fileName(QString card) {
    return QString(":/images/%1.png").arg(card);
}
```

```
QPixmap CardPics::get(QString card) {  
    return m_pixmap[card];  
}  
[ . . . . ]
```

Each resource is used to construct a `QPixmap`, which gets added to `CardPics` for easy access with the `get()` function.

11.5 Layout of Widgets

A widget can be popped up on the screen, like a dialog, or it can be made a part of a larger window. Whenever we wish to arrange smaller widgets inside larger ones, we use *layouts*. A **layout** is an object that belongs to exactly one widget. Its sole responsibility is to organize the space occupied by its owner's child widgets.

Although each widget has a `setGeometry()` function that allows you to set its size and position, absolute sizing and positioning are rarely used in a windowing application because they impose an undesirable rigidity on the design. Proportional resizing, splitters, scrollbars when needed, and flexible arrangement of visual space are all achieved quite naturally through the use of layouts.

The process of specifying the way that your widgets will be arranged on the screen consists of dividing the screen into regions, each controlled by a `QLayout`. Layouts can arrange their widgets

- Vertically (`QVBoxLayout`)
- Horizontally (`QHBoxLayout`)
- In a grid (`QGridLayout`)
- In a stack where only one widget is visible at any time (`QStackedLayout`)

Widgets are added to `QLayouts` using the `addWidget()` function.

Layouts are not widgets, and they have no visual representation. Qt supplies an abstract base class named `QLayout` plus several concrete `QLayout` subclasses: `QBoxLayout` (particularized to `QHBoxLayout` and `QVBoxLayout`), `QGridLayout`, and `QStackedLayout`. Each of the layout types has an appropriate set of functions to control the spacing, sizing, alignment, and access to its widgets.

For its geometry management to work, each `QLayout` object must have a parent that is either a `QWidget` or another `QLayout`. The parent of a `QLayout` can be specified when the layout is constructed by passing the constructor a pointer to the parent widget or layout. It is also possible to construct a `QLayout` without specifying its parent, in which case you can call `QWidget::addLayout()` at some later time.

Layouts can have child layouts. One layout can be added as a sub-layout to another by calling `addLayout()`. The exact signature depends on the kind of layout used. If the parent of a layout is a widget, that widget cannot be the parent of any other layout.

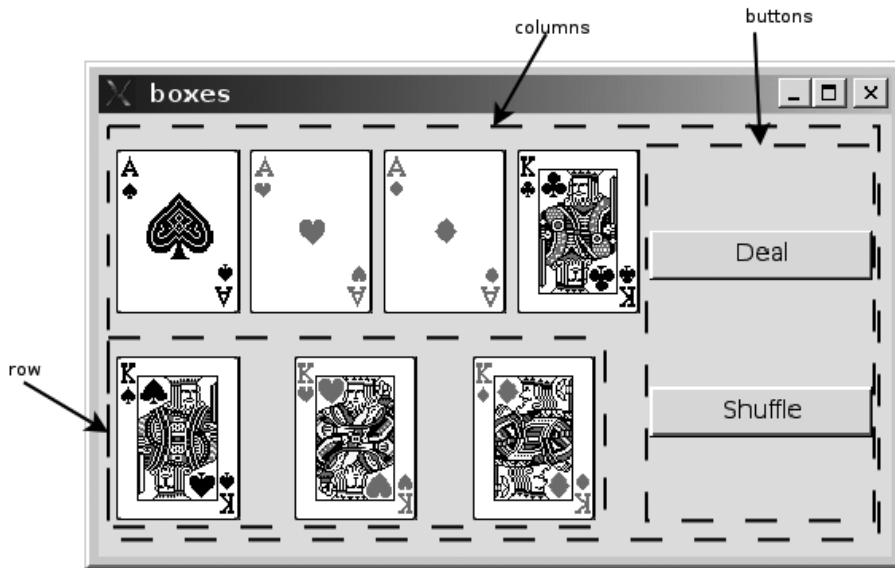


FIGURE 11.5 Rows and columns

The `CardTable` class defined in Example 11.10 reuses `libcards2`, for easy access to `QPixmap`s of playing cards (see Section 11.4). Constructing a `CardTable` object puts Figure 11.5 on the screen.

EXAMPLE 11.10 `src/layouts/boxes/cardtable.h`

```
#ifndef CARDTABLE_H
#define CARDTABLE_H
#include <carddeck.h>
#include <QWidget>

class CardTable : public QWidget {
public:
    CardTable();
private:
    CardDeck m_deck;
};

#endif // #ifndef CARDTABLE_H
```


CardTable is implemented in Example 11.11 by making use of the fact that a QLabel can hold an image. This implementation demonstrates some simple but useful layout techniques.

EXAMPLE 11.11 `src/layouts/boxes/cardtable.cpp`

```
[ . . . . ]
// Given a pixmap, return a label with that pixmap on it.
static QLabel* label(QPixmap pm) {
    QLabel* retval = new QLabel();
    retval->setPixmap(pm);
    return retval;
}

CardTable::CardTable() {

    // create 2 rows of cards:
    QHBoxLayout *row = new QHBoxLayout();
    row->addWidget(label(m_deck.get(1)));
    row->addWidget(label(m_deck.get(2)));
    row->addWidget(label(m_deck.get(3)));
    row->addWidget(label(m_deck.get(4)));

    QVBoxLayout* rows = new QVBoxLayout();
    rows->addLayout(row);

    row = new QHBoxLayout();
    row->addWidget(label(m_deck.get(5)));
    row->addWidget(label(m_deck.get(6)));
    row->addWidget(label(m_deck.get(7)));
    rows->addLayout(row);

    // create a column of buttons:
    QVBoxLayout *buttons = new QVBoxLayout();
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));

    // Bring them together:
    QHBoxLayout* cols = new QHBoxLayout();
    setLayout(cols);           ❶
    cols->addLayout(rows);     ❷
    cols->addLayout(buttons);  ❸
}
[ . . . . ]
```

- ❶ the “root layout” for this widget
- ❷ Add both card rows as a column.
- ❸ Add column of buttons as another column.

The simple piece of client code shown in Example 11.12 suffices to put the window on the screen.

EXAMPLE 11.12 `src/layouts/boxes/boxes.cpp`

```
#include <QApplication>
#include "cardtable.h"

int main(int argc, char* argv[]) {
    QApplication app (argc, argv);
    CardTable ct;
    ct.show();
    return app.exec();
}
```

If you build and run this example and use your mouse to resize the window, you will notice that the width of the buttons stretches first to gobble up extra space, but that there is also stretchable spacing between the cards, as well as between the buttons. If we removed the buttons, we could observe that the horizontal spacing between the cards would grow evenly and uniformly.

11.5.1 Spacing, Stretching, and Struts

To get finer control over the layout of widgets, we can use the API of the `QLayout` class. Box layouts, for example, offer the following functions:

1. `addSpacing(int size)` adds a fixed number of pixels to the end of the layout.
2. `addStretch(int stretch = 0)` adds a stretchable number of pixels. It starts at a minimum amount and stretches to use all available space. In the event of multiple stretches in the same layout, this can be used as a growth factor.
3. `addStrut(int size)` imposes a minimum size to the perpendicular dimension (i.e., the width of a `VBoxLayout` or the height of an `HBoxLayout`).

Revisiting Example 11.11, we will make the layout behave a little better during resizing. Figure 11.6 shows the results of adding some stretch and some spacers to this application.

Normally, layouts try to treat all widgets equally. When we want one widget to be off to a side, or pushed away from another, we can use stretches and spacing to

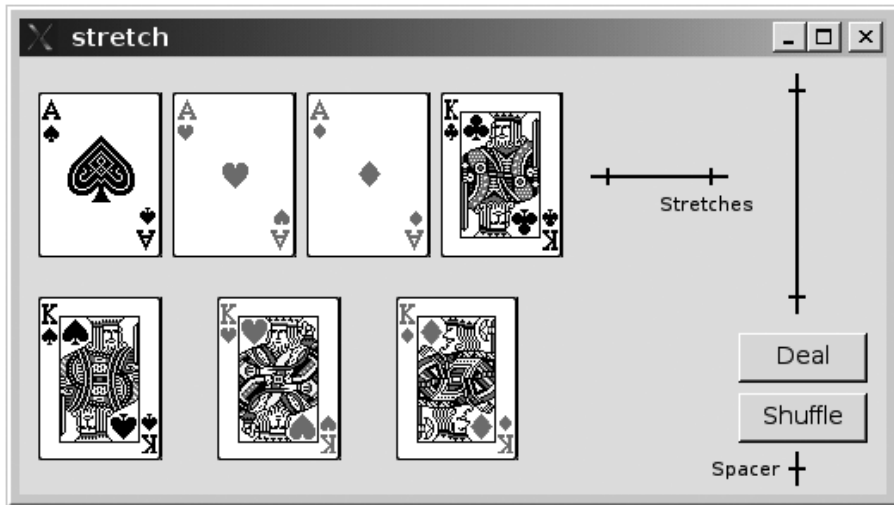


FIGURE 11.6 Improved layout with stretch and spacers

deviate from that norm. Example 11.13 demonstrates how to use stretches and spacing.

EXAMPLE 11.13 `src/layouts/stretch/cardtable.cpp`

```
[ . . . . ]
QVBoxLayout *buttons = new QVBoxLayout();

    buttons->addStretch(1);      ❶
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));
    buttons->addSpacing(20);    ❷

    QHBoxLayout* cols = new QHBoxLayout();
    setLayout(cols);
    cols->addLayout(rows);
    cols->addStretch(30);      ❸
    cols->addLayout(buttons);
}
[ . . . . ]
```

- ❶ stretchable space before buttons in column
- ❷ fixed spacing after buttons
- ❸ Adds a fixed spacing of 30 that stretches

If you build and run this application using the code from Example 11.13 instead of Example 11.11, you can resize the main window and observe that the buttons no longer grow, and are pushed off to the corner. The horizontal spacing between the cards does not grow, but the vertical spacing does.

11.5.2 Moving Widgets across Layouts

Figure 11.7 shows the basic layout for our next example, which demonstrates what happens when a widget is added to more than one layout.

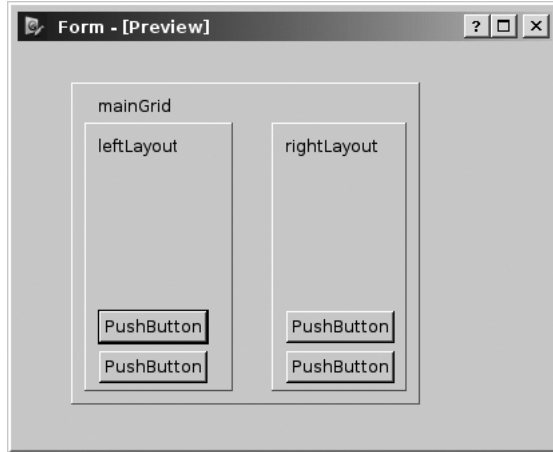


FIGURE 11.7 Moving labels application

This application moves `QLabels` from one layout to the other in response to the button press. In Example 11.14 we derive from `QApplication` a class that defines the GUI.

EXAMPLE 11.14 `src/layouts/moving/moving.h`

```
[ . . . . ]

class MovingApp : public QApplication {
    Q_OBJECT
public:
    MovingApp(int argc, char* argv[]);

public slots:
    void moveLeft();
    void moveRight();
    void newLeft();
    void newRight();
private:
    QString nextLabel();
    QMainWindow m_MainWindow;
    QQueue<QLabel*> m_LeftQueue, m_RightQueue;
    QVBoxLayout *m_LeftLayout, *m_RightLayout;
    int m_Count;
};

[ . . . . ]
```

The constructor starts by creating the layouts and the various widgets, as we see in Example 11.15.

EXAMPLE 11.15 `src/layouts/moving/moving.cpp`

```
[ . . . . ]

MovingApp::MovingApp(int argc, char* argv[]) :
    QApplication(argc, argv),
    m_MainWindow(),
    m_Count(0) {

    QWidget *center = new QWidget(&m_MainWindow);
    m_MainWindow.setCentralWidget(center); ❶

    QGridLayout *mainGrid = new QGridLayout;

    m_LeftLayout = new QVBoxLayout;
    m_RightLayout = new QVBoxLayout;

    mainGrid->addLayout(m_LeftLayout, 0,0);
    mainGrid->addLayout(m_RightLayout, 0, 1);
    QPushButton *moveRight = new QPushButton("Move Right");
    QPushButton *moveLeft = new QPushButton("Move Left");
    mainGrid->addWidget(moveRight, 1,0);
    mainGrid->addWidget(moveLeft, 1,1);

    QPushButton *addRight = new QPushButton("Add Right");
    QPushButton *addLeft = new QPushButton("Add Left");

    mainGrid->addWidget(addLeft, 2,0);
    mainGrid->addWidget(addRight, 2,1);
    center->setLayout(mainGrid);
```

- ❶ The QMainWindow takes ownership of this widget and makes it the central widget. We do not need to delete it.
-

After creation of the various layouts and widgets, signals must be connected to slots, as we see in Example 11.16.

EXAMPLE 11.16 `src/layouts/moving/moving.cpp`

```
[ . . . . ]

connect(moveRight, SIGNAL(pressed()), this, SLOT(moveRight()));
connect(moveLeft, SIGNAL(pressed()), this, SLOT(moveLeft()));
connect(addRight, SIGNAL(pressed()), this, SLOT(newRight()));
connect(addLeft, SIGNAL(pressed()), this, SLOT(newLeft()));
```

continued

```

// What do the insertStretch lines do?
m_LeftLayout->insertStretch(0);
m_RightLayout->insertStretch(0);

newLeft(); ❶
newRight(); ❷
m_MainWindow.move(200,200);
m_MainWindow.resize(300, 500);
m_MainWindow.show();

```

- ❶ This puts a label in the left layout.
- ❷ This puts a label in the right layout.

Because a widget cannot exist in more than one layout at any given time, it disappears from the first layout and shows up in the new one. Each widget retains its parent after the layout change. The code for the movement slots is shown in Example 11.17.

EXAMPLE 11.17 `src/layouts/moving/moving.cpp`

```

[ . . . . ]

void MovingApp::moveLeft() {
    if (m_RightQueue.isEmpty()) return;
    QLabel *top = m_RightQueue.dequeue();
    m_LeftQueue.enqueue(top);
    m_LeftLayout->addWidget(top); ❶
}

void MovingApp::moveRight() {
    if (m_LeftQueue.isEmpty()) return;
    QLabel *top = m_LeftQueue.dequeue();
    m_RightQueue.enqueue(top);
    m_RightLayout->addWidget(top);
}

```

- ❶ By adding it to the left, it disappears from the right.

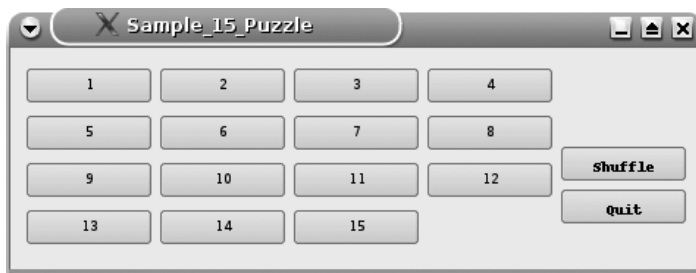
EXERCISES: LAYOUT OF WIDGETS

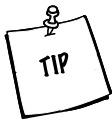
1. There are many ways of getting information from the user. The keeper of the Bridge of Death wants to know the answers to three questions, as we see in the following figure.



- Create a dialog that asks these questions of the user, using `QLineEdit` widgets and two `QPushButton`s to submit or cancel the request. Check the responses to make sure they are correct. If they are not correct, output a funny message.
 - Change the third question randomly so that half of the time it asks “What is the mean air speed velocity of an unladen swallow?”
- 2. The 15 puzzle (or $n^2 - 1$ puzzle) involves a 4×4 ($n \times n$) grid that contains 15 tiles numbered 1 to 15 (1 to $n^2 - 1$), and one empty space. The only tiles that can move are those next to the empty space.
 - Create a 15 puzzle with `QPushButton`s in a `QGridLayout`.
 - At the start of the game, the tiles are presented to the player in “random” order. The object of the game is to rearrange them so that they are in ascending order, with the lowest numbered tile in the upper-left corner.
 - If the player solves the puzzle, pop up a `QMessageBox` saying “YOU WIN!” (or something more clever).
 - Add some buttons:
 - Shuffle: Randomize the tiles by performing a large number (at least 50) of legal tile slides.
 - Quit: To leave the game.

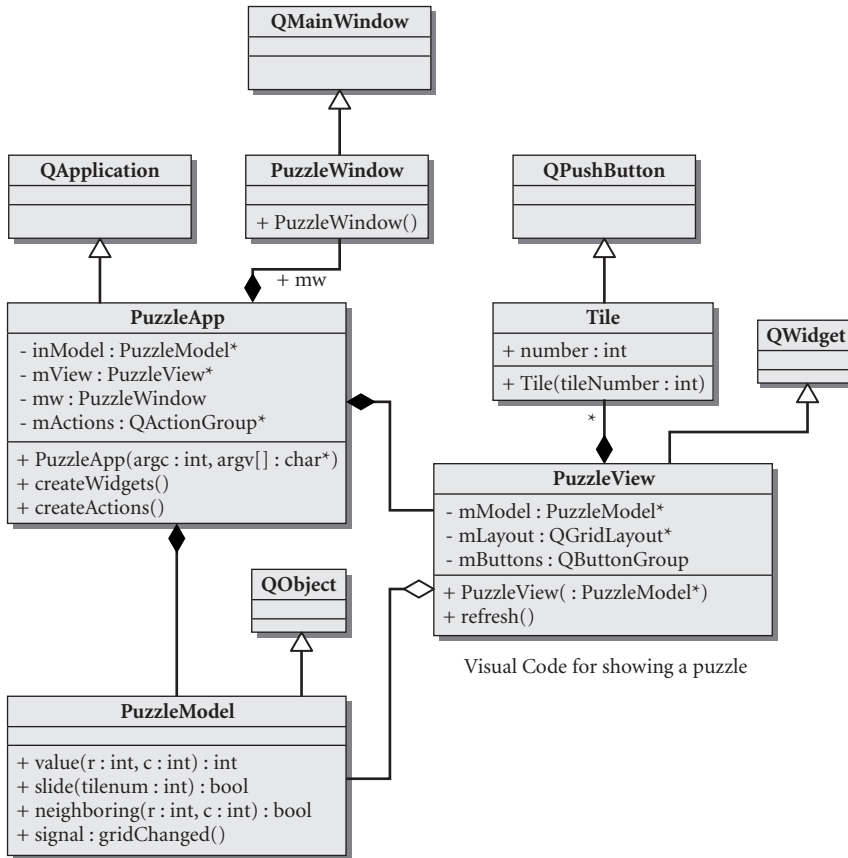
Sample 15 Puzzle





DESIGN SUGGESTIONS Want to get a head start using the model-view-controller style? It comes up later, but you can try it now. Define the classes shown in the accompanying figure and try to partition your code properly into them.

Model-View-Controller Design for Puzzle



Non-Visual class - for representing the “state” of the puzzle

11.6 QAction, QMenu, and QMenuBar

A `QAction` is a `QObject` that is a base class for user-selected actions. It provides a rich interface that can be used for a wide variety of actions, as we will soon see. The `QWidget` interface enables each widget to maintain a `QList<QAction*>`.

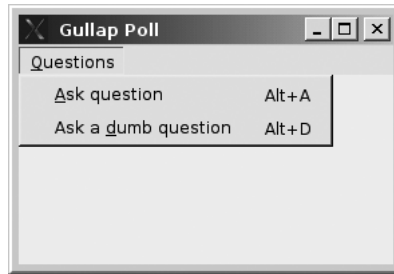
A `QMenu` is a `QWidget` that provides a particular kind of view for a collection of `QActions`. A `QMenuBar` is a collection of menus.

When the parent of a `QMenu` is a `QMenuBar`, the `QMenu` appears as a pull-down menu with a familiar interface. When its parent is not a `QMenuBar` it can pop up, like a dialog, in which case it is considered a **context menu**.² A `QMenu` can have another `QMenu` as its parent, in which case it becomes a submenu.

To help the user make the right choice, each action can have the following:

- Text and/or icon that appears on a menu and/or button
- An accelerator or a shortcut key
- A “What’s this?” and a tool-tip
- A way to toggle the state of the action between visible/invisible, enabled/disabled, and checked/not checked
- `changed()`, `hovered()`, `toggled()`, and `triggered()` signals

The `Dialog` in Example 11.4 had a menubar with a single menu that gave two choices.



Example 11.18 shows the code that sets up that menubar.

EXAMPLE 11.18 `src/widgets/dialogs/messagebox/dialogs.cpp`

```
[ . . . . ]

/* Insert a menu into the menubar */
QMenu *menu = new QMenu("&Questions", this);

QMainWindow::menuBar()->addMenu(menu);

/* Add some choices to the menu */
menu->addAction("&Ask question",
               this, SLOT(askQuestion()), tr("Alt+A"));
menu->addAction("Ask a &dumb question",
               this, SLOT(askDumbQuestion()), tr("Alt+D"));
}
```

² A context menu is usually activated by clicking the right mouse button or by pressing the “menu” button. It is called a context menu because the menu always depends on the context (which `QWidget` is currently selected or focused).

The calls to `QMenu::addAction(text, target, slot, shortcut)` each create an unnamed `QAction` and call `QWidget::addAction(QAction*)` to install it in the menu. The latter call adds the new action to the menu's `QList<QAction*>`.

11.7 QActions, QToolbars, and QActionGroups

11.7.1 The Command Pattern

The **Command pattern**, as described in [Gamma95] encapsulates operations as objects with a common execution interface. This can make it possible to place operations in a queue, log operations, and undo the results of an already executed operation.

Because an application might provide a variety of different ways for the user to issue the same command (e.g., menus, toolbar buttons, keyboard shortcuts), encapsulating each command as an *action* helps to ensure consistent, synchronized behavior across the application. `QAction` is, therefore, an ideal base class for implementing the Command pattern.

In Qt GUI applications, actions are typically “triggered” in one of the following ways:

- A user clicks on a menu choice.
- A user presses a shortcut key.
- A user clicks on a toolbar button.

There are several overloaded forms of `QMenu::addAction()`. We will use the version inherited from `QWidget`, `addAction(QAction*)` in Example 11.19. Here we see how to add actions to menus, action groups, and toolbars. We start by deriving a class from `QMainWindow` and equipping it with several `QAction` members plus a `QActionGroup` and a `QToolBar`.

EXAMPLE 11.19 `src/widgets/menus/study.h`

```
[ . . . . ]
class Study : public QMainWindow {
    Q_OBJECT
public:
    Study();
```

```

public slots:
    void actionEvent(QAction* act);
private:

    QActionGroup* actionGroup; ❶
    QToolBar *toolbar;         ❷

    QAction *useTheForce;
    QAction *useTheDarkSide;
    QAction *studyWithObiWan;
    QAction *studyWithYoda;
    QAction *studyWithEmperor;
    QAction *fightYoda;
    QAction *fightDarthVader;
    QAction *fightObiWan;
    QAction *fightEmperor;

protected:
    QAction* addChoice(QString name, QString text);

};
[ . . . . ]

```

- ❶ for catching the signals
- ❷ for displaying the actions as buttons

The constructor for this class sets up the menus and installs them in the `QMenuBar` that is already part of the base class. (See Example 11.20.)

EXAMPLE 11.20 `src/widgets/menus/study.cpp`

```

[. . . . ]

Study::Study() {
    actionGroup = new QActionGroup(this);
    actionGroup->setExclusive(false);
    statusBar();

    QWidget::setWindowTitle( "to become a jedi, you wish?" ); ❶
    QMenu* useMenu = new QMenu("&Use", this);
    QMenu* studyMenu = new QMenu("&Study", this);
    QMenu* fightMenu = new QMenu("&Fight", this);

    useTheForce = addChoice("useTheForce", "Use The &Force");
    useTheForce->setStatusTip("This is the start of a
    journey...");
    useTheForce->setEnabled(true);
    useMenu->addAction(useTheForce); ❷

    [ . . . . ]

```

continued

```

studyWithObiWan = addChoice("studyWithObiWan", "&Study With
Obi Wan");
studyMenu->addAction(studyWithObiWan);
studyWithObiWan->setStatusTip("He will certainly open doors
for you...");

fightObiWan = addChoice("fightObiWan", "Fight &Obi Wan");
fightMenu->addAction(fightObiWan);
fightObiWan->setStatusTip(
    "You'll learn some tricks from him that way, for sure!");
[ . . . . ]

QMainWindow::menuBar()->addMenu(useMenu);
QMainWindow::menuBar()->addMenu(studyMenu);
QMainWindow::menuBar()->addMenu(fightMenu);

toolbar = new QToolBar("Choice ToolBar", this);
toolbar->addAction(actionGroup->actions());

QMainWindow::addToolBar(Qt::LeftToolBarArea, toolbar);

QObject::connect(actionGroup, SIGNAL(triggered(QAction*)),
    this, SLOT(actionEvent(QAction*)));

QWidget::move(300, 300);
QWidget::resize(300, 300);
}

```

- ❶ The ClassName:: prefixes we use in methods here are not necessary, because the methods can be called on "this". We list the classname only to show the human reader from which class the method was inherited.
- ❷ It's already in a QActionGroup, but we also add it to a QMenu.
- ❸ This gives us visible buttons in a dockable widget for each of the QActions.
- ❹ Instead of connecting each individual action's signal, we perform one connect to an actionGroup that contains them all.

It is possible to connect individual `QAction` `triggered()` signals to individual slots. It is also possible to group related `QActions` together in a `QActionGroup`, as we have just done. `QActionGroup` offers a single signal `triggered(QAction*)`, which makes it possible to handle the group of actions in a uniform way.

After being created, each `QAction` is added to three other objects (via `addAction()`):

1. A `QActionGroup`, for signal handling
2. A `QMenu`, one of three possible pull-down menus in a `QMenuBar`
3. A `QToolBar`, where it is rendered as a button

EXAMPLE 11.21 src/widgets/menus/study.cpp

```
[ . . . . ]

// Factory method for creating QAction's initialized in a uniform way
QAction* Study::addChoice(QString name, QString text) {
    QAction* retval = new QAction(text, this);
    retval->setObjectName(name);
    retval->setEnabled(false);
    retval->setCheckable(true);
    actionGroup->addAction(retval); ❶
    return retval;
}

```

❶ Add every action we create to a QActionGroup so that we only connect one signal to a slot.

To make this example a bit more interesting, we established some logical dependencies between the menu choices so that they were consistent with the plot of the various movies. This logic is expressed in the `actionEvent()` function. (See Example 11.22).

EXAMPLE 11.22 src/widgets/menus/study.cpp

```
[ . . . . ]

void Study::actionEvent(QAction* act) {
    QString name = act->objectName();
    QString msg = QString();

    if (act == useTheForce ) {
        studyWithObiWan->setEnabled(true);
        fightObiWan->setEnabled(true);
        useTheDarkSide->setEnabled(true);
    }
    if (act == useTheDarkSide) {
        studyWithYoda->setEnabled(false);
        fightYoda->setEnabled(true);
        studyWithEmperor->setEnabled(true);
        fightEmperor->setEnabled(true);
        fightDarthVader->setEnabled(true);
    }

    if (act == studyWithObiWan) {
        fightObiWan->setEnabled(true);
        fightDarthVader->setEnabled(true);
        studyWithYoda->setEnabled(true);
    }
}
[ . . . . ]

```

continued

```

    if (act == fightObiWan ) {
        if (studyWithEmperor->isChecked()) {
            msg = "You are victorious!";
        }
        else {
            msg = "You lose.";
            act->setChecked(false);
            studyWithYoda->setEnabled(false);
        }
    }
    [ . . . . ]

    if (msg != QString()) {
        QMessageBox::information(this, "Result", msg, "ok");
    }
}

```

Because all actions are in a `QActionGroup`, a single `triggered(QAction*)` signal can be connected to the `actionEvent()` slot.

The client code in Example 11.23 shows how the program starts.

EXAMPLE 11.23 `src/widgets/menus/study.cpp`

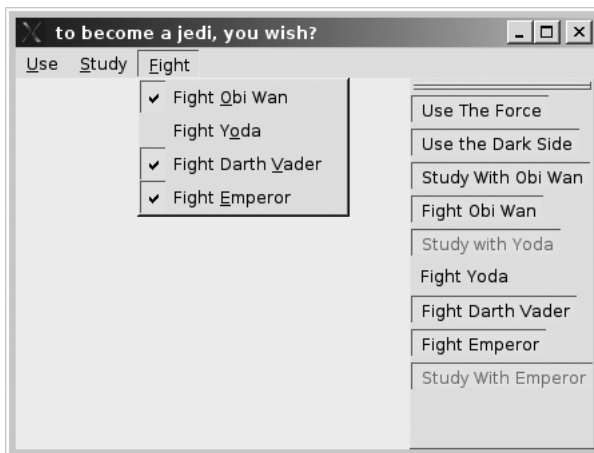
```

[ . . . . ]

int main( int argc, char ** argv ) {
    QApplication a( argc, argv );
    Study study;
    study.show();
    return a.exec();
}

```

Here is a screenshot of the running program.



All menu choices except one are initially disabled. As the user selects from the available choices, other options become enabled or disabled. Also, notice that there is consistency between the buttons and the choices in the menus. Clicking on an enabled button causes the corresponding menu item to be checked. `QAction` stores the state (enabled/checked), and the `QMenu` and `QToolBar` provide views of the `QAction`.

EXERCISES: QActions, QMenus, AND QMenuBarS

1. (Discussion question) There are `QActions` as children of `QWidget`s all over the place. How do you gather them all for a `ShortcutView` widget that lets the user display and change all of the keyboard shortcuts in the application?
2. Revisit the 15 puzzle application in Section 11.5.2 and add `QActions` for:
 - Shuffle puzzle
 - Reset puzzle
 - Quit

For Quit, pop up a message box asking whether the user is sure before actually quitting.

3. Write a “login” application using `QMainWindow`. Start with these `QActions`:
 - Login
 - Create New User
 - Edit Preferences
 - Change Password

These choices should be available by pull-down menu as well as toolbar. Create `QActions` and a `QActionGroup` for them all.

The last two choices should be disabled unless the user is logged in.

The initial login screen should have `QLineEdit`s for a user id and a password. If the user chooses “Create New User” or “Change Password”, the program should display a new form with `QLineEdit`s. It should ask for the password twice, and make sure the passwords match, before actually performing the operation. If the passwords mismatch, try again.

For “Login”, it should check that the user exists and that the password is valid, before letting you log in, and enabling the other `QActions`.

If the user chooses “Edit Preferences” it should ask the following questions:

- What is your name?
- What is your quest?

- What is your favorite color? (Use a `QComboBox` with preset colors to red, green, blue, black, pink, and chartreuse to choose from.)

It should remember these for each user from previous sessions.

The program should load the user data from a file called “users.xml” on startup and save to the same file when finished, without any interaction. Load and save in any format you want, using `QTextStream` and `QFile`.

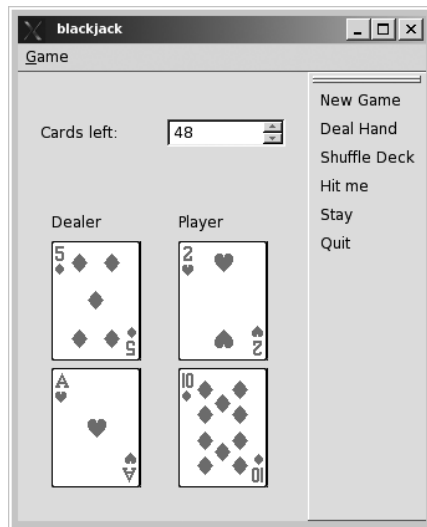
EXERCISE: CARD GAME GUI

Write a blackjack game, with the following actions:

1. New game
2. Deal new hand
3. Shuffle deck
4. Hit me—ask for another card
5. Stay—evaluate my hand
6. Quit—stop playing

These actions and the rules of the game are explained below.

When the game starts, the user and the dealer are each dealt a “hand” of cards. Each hand initially consists of two cards. The user plays her hand first by deciding to add cards to her hand with the “Hit me” action zero or more times. Each Hit adds one card to her hand. The user signals that she wants no more cards with the “Stay” action.



For the purposes of evaluation of a hand, a “face card” (Jack, Queen, and King) counts as 10 points, an Ace can count as 1 or 11 points, whichever is best. Each other card has a number and a point value equal to that number. If the hand consists of an Ace plus a Jack, then it is best to count the Ace as 11 so that the total score is 21. But if the hand consists of an 8 plus a 7, and an Ace is added to the hand, it is best to count the Ace as 1.

The object of the game is to achieve the highest point total that is not greater than 21. If a player gets a point total greater than 21, that player is “busted” (loses) and the hand is finished.

If a player gets five cards in her hand with a total that is not greater than 21, then that player wins the hand.

After the user either wins, loses, or Stays, the dealer can take as many hits as necessary to obtain a point total greater than 18. When that state is reached the dealer must Stay and the hand is finished. The player whose score is closer to, but not greater than, 21 wins. If the two scores are equal, the dealer wins.

When the hand is over, the user can only select “Deal hand”, “New game”, or “Quit” (i.e., Hit and Stay are disabled).

After the user selects “Deal hand”, that choice should be disabled until the hand is finished.

Keep track of the number of games won by the dealer and by the user, starting with zero for each player, by adding one point to the total for the winner of each hand. Display these totals above the cards.

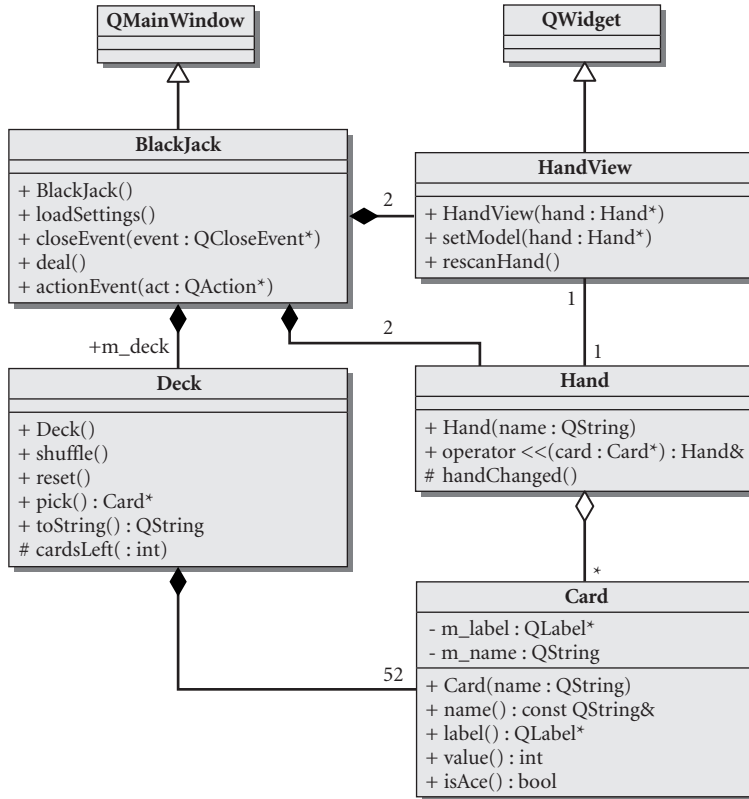
Deal more cards without resetting the deck until the deck becomes empty or the user chooses “Shuffle deck”. Try to reuse or extend the `CardDeck` and related classes that you developed earlier in Section 10.3. Add a graphical representation to your game by showing a `QLabel` with a `QPixmap` for each card, as was done in Chapter 11.

Provide a pull-down menu and a toolbar for each of the `QActions`. Make sure that Hit and Stay are only enabled after the game has started.

Show how many cards are left in the deck in a read only `QSpinBox` at the top of the window.

New game should zero the games won totals and reset the deck.

BlackJack UML Diagram



11.7.1.1 Design Suggestions

Try to keep the model classes separate from the view classes, rather than adding GUI to the model classes. Keeping a strong separation between model and view will give us benefits later.

Figure 11.10 is only a starting point. You need to decide on the base class(es) to extend for defining the model classes, as well as which containers to reuse.

To modify the UML diagram you can try loading the diagram file, `cardgame.xmi`, into `umbrello`.

11.8 Regions and QDockWidgets

Any class that derives from `QMainWindow` has four dock window regions, one on each of the four sides of the central widget. These four regions are used for attaching secondary windows to the central widget.

A `QDockWidget` can be thought of as an envelope for another widget. It has a title bar and a content area to contain the other widget. Depending on how it is set, a `QDockWidget` can be undocked, resized, dragged to a different location, or docked to the same or to a different dock window region by the end user.

The `QMainWindow` correctly creates the slidable `QSplitters` between the central widget and the `QDockWidgets`. The two principal `QMainWindow` functions for managing the dock window regions are

1. `setCentralWidget (QWidget*)`, which establishes the central widget
2. `addDockWidget (Qt::DockWidgetAreas, QDockWidget*)`, which adds the given `QDockWidget` to the specified dock window region.

`DockWindows` are very important in integrated development environments, because different tools or views are needed in different situations. Each view is a widget, and they can all be “plugged” into the main window quite easily with the docking mechanism as shown in Figure 11.8.

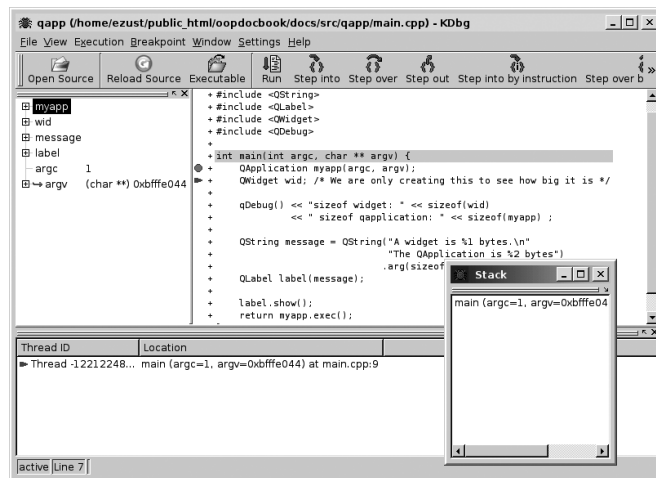


FIGURE 11.8 Dragged DockWindows

Like most KDE applications, `kdbg`, the stand-alone KDE debugger, makes use of `DockWindows`. `kdbg` has a source code window (central widget) and dockable views for things such as:

- Program stack
- Output
- Watches
- Local variables
- Threads

These widgets do not all need to be visible at the same time, so there is a View menu that lets you select or de-select all of the different views. `QMainWindow::createPopupMenu()` returns this menu, permitting you to add it to toolbars or other pull-down menus.

11.9 Views of a QStringList

Example 11.24 demonstrates the use of the simplest concrete model, `QStringListModel`.

EXAMPLE 11.24 `src/modelview/qstringlistmodel/simplelistapp.h`

```
#ifndef SIMPLELIST_H
#define SIMPLELIST_H

#include <QApplication>
#include <QListView>
#include <QStringListModel>
#include <QMainWindow>
#include <QPushButton>
/* Controller example */

class SimpleListApp : public QApplication {
    Q_OBJECT
public:
    SimpleListApp(int argc, char* argv[]);
public slots:
    void showNewChanges();
    void addItem();
private:
    QStringListModel m_Model;    ❶
    QMainWindow m_Window;
    QListView m_View;
    QPushButton m_Button;      ❷
};

#endif
```

- ❶ created first
- ❷ destroyed first

`SimpleListApp` extends `QApplication` and is the “managing object” for all others.

Because all of the objects are direct children of the `SimpleListApp`, we defined the other objects as members, rather than pointers to heap objects. Their creation and destruction is then done during `SimpleListApp`’s creation and destruction. The model is listed before the view so that it is destroyed after the view.

The implementation of this class, with a bit of client code, is shown in Example 11.25.

EXAMPLE 11.25 `src/modelview/qstringlistmodel/simplelistapp.cpp`

```
#include "simplelistapp.h"
#include <QDebug>
#include <QVBoxLayout>

SimpleListApp::SimpleListApp(int argc, char* argv[] ) :
    QApplication(argc, argv), m_Button("Insert") {
    QString englishDays = "Monday,Tuesday,Wednesday,Thursday,Friday,"
        "Saturday,Sunday";
    QString frenchDays = "Lundi,Mardi,Mercredi,Jeudi,Vendredi"
        ",Samedi,Dimanche";
    QString dutchDays = "Mandaag,Dinsdag,Wowoonsdag,Dunderdag,"
        "Vrijdag,Zaterdag,Zonedaag";
    QStringList days = dutchDays.split(",");
    m_Model.setStringList(days);
    m_View.setModel(&m_Model);
    connect(this, SIGNAL(aboutToQuit()),
        this, SLOT(showNewChanges()));
    QWidget *wid = new QWidget(&m_Window);
    QVBoxLayout *layout = new QVBoxLayout(wid);
    m_Window.setCentralWidget(wid);
    layout->addWidget(&m_View);
    layout->addWidget(&m_Button);
    connect (&m_Button, SIGNAL(clicked()),
        this, SLOT(addItem()));
    m_Window.setVisible(true);
}

void SimpleListApp::addItem() {
    static int itemnumber = 1;
    QString str = QString("item #%1").arg(itemnumber++);
    QStringList sl = m_Model.stringList();
    sl << str;
    m_Model.setStringList(sl);
}

void SimpleListApp::showNewChanges() {
    qDebug() << " The new days of the week: ";
    qDebug() << m_Model.stringList().join(",");
}

int main(int argc, char* argv[]) {
    SimpleListApp app(argc, argv);
    return app.exec();
}
```

When the app in Example 11.24 is run an editable `QListView` appears. You can use the mouse or arrow keys to navigate, and use F2 or double-click to edit. After you exit, you will see the revised days of the week:

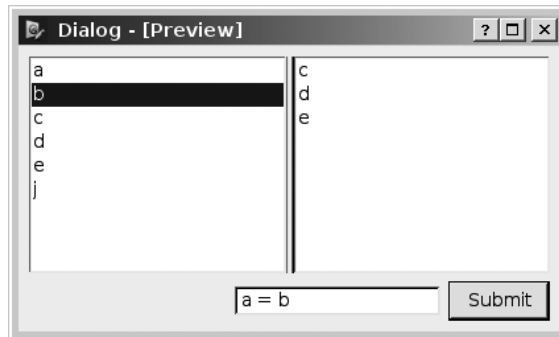


The new days of the week:

"Mandaag, Dinsdag, Wednesday, Thursday, Friday, Saturday, Zonedag"

EXERCISE: VIEWS OF A QSTRINGLIST

Provide a graphical front-end for the equivalence engine from Section 10.7. The UI should have a `QLineEdit` for the the user to type in new equivalences, and two `QListView` widgets side by side above it.



In the diagram shown here, the item `b` is selected on the left, and we can see that `b`'s equivalences are `c`, `d`, `e` on the right. Any time the "Submit" button is clicked, whatever is in the `TextEditor` is added to the equivalence engine, and we should be able to see that in subsequent views.

POINTS OF DEPARTURE

- Read the Qt Designer Manual and follow the tutorial to familiarize yourself with a more graphical way of defining GUIs.
- *Qt Quarterly*³ has an article that discusses how to write a shortcut editor.

³ <http://doc.trolltech.com/qq/qq14-actioneditor.html>

REVIEW QUESTIONS

1. List six things that `QWidget`s have in common.
2. How can you save and later restore the size, position, and arrangements of widgets for a GUI app?
3. Why would you want to do such a thing?
4. What is a dialog? Where is an appropriate place to use it?
5. What is a `QLayout`? What is its purpose? What is an example of a concrete `QLayout` class?
6. Can a widget be a child of a layout?
7. Can a layout be a child of a widget?
8. Revisiting Example 11.15, how does the `QGridLayout` determine the number of columns? What would be the effect of using

```
mainGrid->addWidget(moveLeft, 1,2);
```

in line 25?
9. What are the advantages of listing our images in a resources file?
10. What is the difference between a spacer and a stretch?
11. What is a `QAction`? How are actions triggered?
12. It is possible to create `QMenus` without using `QActions`. What are the advantages of using a `QAction`?

12

CHAPTER 12

Concurrency

`QProcess` and `QThread` provide two approaches to concurrency. We discuss how to create and communicate with processes and threads, as well as techniques for monitoring and debugging them.

12.1	QProcess and Process Control.....	278
12.2	Threads and QThread	290
12.3	Summary: QProcess and QThread....	303

12.1 QProcess and Process Control

`QProcess` is a very convenient (and cross-platform) class for starting and controlling other processes. It is derived from `QObject` and takes full advantage of signals and slots to make it easier to “hook up” with other Qt classes.

We discuss now a simple example that starts a process and views its continually running output. Example 12.1 shows the definition of a simple class derived from `QProcess`.¹

EXAMPLE 12.1 `src/logtail/logtail.h`

```
[ . . . . ]
#include <QObject>
#include <QProcess>
class LogTail : public QProcess {
    Q_OBJECT
public:
    LogTail(QString fn = QString());
    ~LogTail();
public slots:
    void logOutput();
};
[ . . . . ]
```

A `QProcess` can spawn another process using the `start()` function. The new process is a child process in that it will terminate when the parent process does. Example 12.2 shows the implementation of the constructor and destructor of the `LogTail` class.²

EXAMPLE 12.2 `src/logtail/logtail.cpp`

```
[ . . . . ]

LogTail::LogTail(QString fn) {
    if (fn == QString()) {
        fn = "/var/log/apache/access.log";
    }
}
```

¹ `tail -f` runs forever showing whatever is appended to a file, and is useful for showing the contents of a log file of a running process.

² It is also possible to use `startDetached()` to start a process that will continue to live after the calling process exits.

```

    connect (this, SIGNAL(readyReadStandardOutput()),
            this, SLOT(logOutput())); ❶
    QStringList argv;
    argv << "-f" << fn;
    // We want to exec "tail -f filename"
    start("tail", argv); ❷
}

LogTail::~LogTail() {
    terminate(); ❸
}

```

- ❶ When there is input ready, we will know about it.
- ❷ Returns immediately, and now there is a forked process running independently but “attached” to this process. When the calling process exits, the forked process will terminate.
- ❸ Attempts to terminate this process.

The child process can be treated as a sequential I/O device with two predefined output channels that represent two separate streams of data: `stdout` and `stderr`. The parent process can select an output channel with `setReadChannel()` (default is `stdout`). The signal `readyReadStandardOutput()` is emitted when data is available on the selected channel of the child process. The parent process can then read its output by calling `read()`, `readLine()`, or `getChar()`. If the child process has standard input enabled, the parent can use `write()` to send data to it.

Example 12.3 shows the implementation of the slot `logOutput()`, which is connected to the signal `readyReadStandardOutput()` and uses `readAllStandardOutput()` so that it pays attention only to `stdout`.

EXAMPLE 12.3 `src/logtail/logtail.cpp`

```

[ . . . . ]

// tail sends its output to stdout.
void LogTail::logOutput() { ❶
    QByteArray bytes = readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        qDebug() << line;
    }
}

```

- ❶ event driven—passive interface

The use of signals eliminates the need for a read loop. When there is no more input to be read, the slot will no longer be called. Signals and slots make concurrent code much simpler to read, because they hide the event-handling and dispatching code. Some client code is shown in Example 12.4.

EXAMPLE 12.4 `src/logtail/logtail.cpp`

```
[ . . . . ]

#include <QApplication>
#include <logwindow.h>
#include <argumentlist.h>

int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
    ArgumentList al;
    LogWindow lw("debug"); ❶
    lw.setWindowTitle("logtail demo");
    QString filename;
    if (al.size() > 1) filename = al[1];
    LogTail tail(filename); ❷
    lw.show();
    return app.exec();
}
```

- ❶ Create a scrolling edit window watching debug messages.
- ❷ Create object, but start process too.

This application appends lines to the `LogWindow` whenever they appear in the specified log file. The default log file is named `access.log` for the apache Web server on a typical *nix local host.

12.1.1 Processes and Environment

Environment variables are name/value string pairs that can be stored quite easily in a map or a hash table. Every running process has an **environment**, or a collection of environment variables. Most programming languages support a way of getting and setting these variables.

The most common environment variables are

1. `PATH`, a list of directories to search for executables
2. `HOME`, the location of your home directory
3. `HOSTNAME` (*nix) or `COMPUTERNAME` (win32), which usually gives you the name of your machine
4. `USER` (*nix) or `USERNAME` (Win32), which usually gives you the currently logged-in user

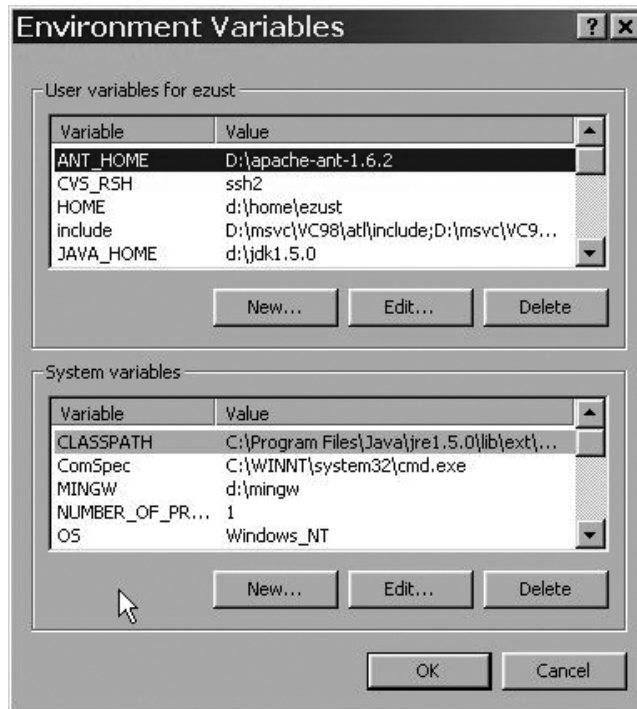
Another environment variable you will encounter in the context of the exercises is `CPPLIBS`, which denotes the location of our own C++ libraries.

Environment variables and their values are arbitrarily set by the parent process, so they can be any pair of strings. Programs that depend on specific variables are

generally not portable, but depend somehow on their parent process. Variables provide a convenient *cross-language* mechanism for communicating information between processes.

Operating system shells allow the user to set environment variables for that process and its future children. Here are some examples.

1. Microsoft Windows desktop: start -> settings -> system -> advanced -> environment variables



2. Microsoft command prompt: `set VARIABLE=value` and `echo %VARIABLE%`
3. bash command line: `export VARIABLE=value` and `echo $VARIABLE`

Many programming languages support getting and setting environment variables too, as listed here.

1. C/C++: `getenv()` and `putenv()` from `<cstdlib>` (see Appendix B)³
2. Python: `os.getenv()` and `os.putenv()`
3. Perl: `%ENV` hashtable
4. Java 1.5: `ProcessBuilder.environment()`
5. Qt 4: `QProcess::environment()`

³ Some platforms also offer `setenv()`, which is more convenient but less portable.

Changing the environment does not affect other processes that are already running. Environments are inherited from the parent process at process creation time.

Any program that you run is quite possibly a few levels deep in the process tree. This is because the typical desktop operating system environment consists of many processes running together. In Figure 12.1 indentation levels indicate the parent-child relationships. Processes that are at the same indentation level are siblings (i.e., children of the same parent).

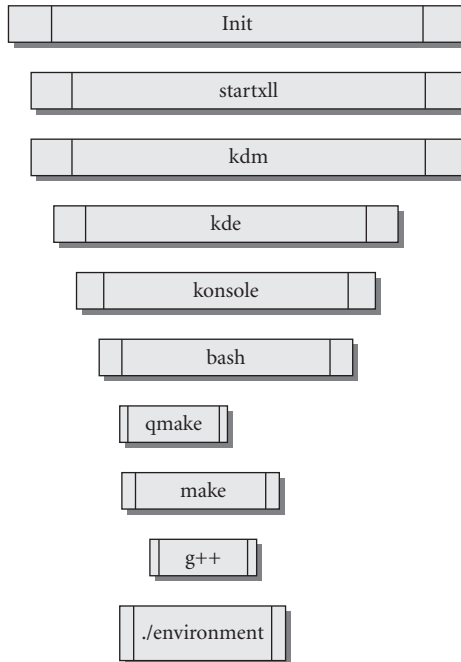


FIGURE 12.1 Linux process hierarchy

Whenever program A runs program B, A is the parent process of B. B inherits (a copy of) A's environment when B is created. Changes to B's environment from inside B will only affect B and B's future children, and will always be invisible to A.

In Example 12.5, we verify that the value given to `setenv()` is propagated to its child process.

EXAMPLE 12.5 `src/environment/setenv.cpp`

```

#include <qstd.h>
#include <argumentlist.h>
#include <QProcess>
#include <cstdlib>

```

```

class Fork : public QProcess {
public:
    Fork(QStringList argv = QStringList() ) {
        execute("environment", argv); ❶
    }
    ~Fork() {
        waitForFinished();
    }
};

int main(int argc, char* argv[]) {
    using namespace qstd;
    ArgumentList al(argc, argv);

    al.removeFirst();
    bool fork=al.getSwitch("-f");

    QStringList extraVars;
    if (al.count() > 0) {
        setenv("PENGUIN", al.first().toAscii(), true);
    }
    cout << " HOME=" << getenv("HOME") << endl;
    cout << " PWD=" << getenv("PWD") << endl;
    cout << " PENGUIN=" << getenv ("PENGUIN") << endl;

    if (fork) {
        Fork f;
    }
}

```

❶ Runs this same app as a child.

When this program is run, our output looks like this:

```

/home/lazarus/src/environment> export PENGUIN=tux
/home/lazarus/src/environment> ./environment -f
HOME=/home/lazarus
PWD=/home/lazarus/src/environment
PENGUIN=tux
HOME=/home/lazarus
PWD=/home/lazarus/src/environment
PENGUIN=tux
/home/lazarus/src/environment> ./environment -f opus
HOME=/home/lazarus
PWD=/home/lazarus/src/environment
PENGUIN=opus
HOME=/home/lazarus
PWD=/home/lazarus/src/environment
PENGUIN=opus

```

12.1.2 Qonsole: Writing an Xterm in Qt

A command line shell reads commands from the user and prints the output. In this example, we have a `LogWindow` (see Section 7.4) providing a **view** of the output of another running process, in this case, `bash`. The `QProcess` is a **model**, representing a running process. Figure 12.2 shows a screenshot of `Qonsole`, our first attempt at a GUI for a command shell.



FIGURE 12.2 Qonsole1

Because it connects signals to slots and handles user interactions, `Qonsole` is considered a **controller**. Because it derives from `QMainWindow`, it also contains some view code. The UML diagram in Figure 12.3 shows the relationships between the classes in this application.

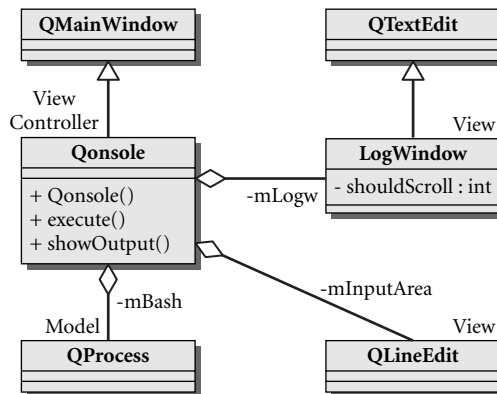


FIGURE 12.3 Qonsole UML: Model and view

In Example 12.6 we see how the constructor establishes the structure of `Qonsole` and the important connections between its components.

EXAMPLE 12.6 src/qconsole/qconsole1/qconsole.cpp

```
[ . . . . ]

Qconsole::Qconsole() {
    m_Logw = new LogWindow("debug");
    m_Logw->setReadOnly(true);
    setCentralWidget(m_Logw);
    m_InputArea = new QLineEdit();
    QDockWidget* qdw = new QDockWidget("Type commands here");
    qdw->setWidget(m_InputArea);
    addDockWidget(Qt::BottomDockWidgetArea, qdw);
    connect (m_InputArea, SIGNAL(returnPressed()),
            this, SLOT(execute()));
    m_Bash = new QProcess();
    m_Bash->setReadChannelMode(QProcess::MergedChannels); ❶
    connect (m_Bash, SIGNAL(readyReadStandardOutput()),
            this, SLOT(showOutput()));
    m_Bash->start("bash", QStringList() << "-i"); ❷
}

```

- ❶ Merge stdout and stderr.
- ❷ Run bash in interactive mode.

Whenever `bash` outputs anything, `Qconsole` sends it to the `LogWindow`. Whenever the user presses the return key, `Qconsole` grabs any text that is in the `QLineEdit` and sends it to `bash`, which interprets it as a command, as we see in Example 12.7.

EXAMPLE 12.7 src/qconsole/qconsole1/qconsole.cpp

```
[ . . . . ]

void Qconsole::showOutput() { ❶
    QByteArray bytes = m_Bash->readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        m_Logw->append(line);
    }
}

void Qconsole::execute() {
    QString cmdStr = m_InputArea->text() + "\n";
    m_InputArea->setText("");
    m_Logw->append(cmdStr);
    QByteArray bytes = cmdStr.toUtf8(); ❷
    m_Bash->write(bytes); ❸
}

```

- ❶ A slot that gets called whenever input is ready
- ❷ 8-bit Unicode Transformation Format
- ❸ Send the data into the stdin stream of the bash child process

Example 12.8 shows the client code that launches this application.

EXAMPLE 12.8 `src/qonsole/qonsole1/qonsole.cpp`

```
[ . . . . ]

#include <QApplication>

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    Qonsole qon;
    qon.show();
    return app.exec();
}
```

12.1.3 Qonsole with Keyboard Events

In the preceding example, Qonsole had a separate widget for user input. For a more authentic xterm experience, the user should be able to type commands in the command output window. To accomplish this Qonsole needs to capture keyboard events. The first step is to override the QObject base class `eventFilter()` method, as we see in Example 12.9, the revised class definition.

EXAMPLE 12.9 `src/qonsole/keyevents/qonsole.h`

```
[ . . . . ]
class Qonsole : public QMainWindow {
    Q_OBJECT
public:
    Qonsole();
    public slots:
        void execute();
        void showOutput();
        bool eventFilter(QObject *o, QEvent *e) ;
protected:
    void updateCursor();
private:
    QString m_UserInput;
    LogWindow* m_Logw;
    QProcess* m_Bash;
};
[ . . . . ]
```

As we discussed in Section 9.3, an event is an object derived from `QEvent`. Within the context of an application, such a `QEvent` is associated with a `QObject` that is its intended receiver. The receiving object has a handler to process the event. An *eventFilter* examines a `QEvent` and determines whether or not to permit it to be processed by the intended receiver. We have provided our revised Qonsole application with an `eventFilter()` function that will be used to filter keyboard

events from `m_Logw`, an extended `QTextEdit`. `QOnsole`, an extended `QMainWindow`, is the intended recipient of those events. The implementation of this function is shown in Example 12.10.

EXAMPLE 12.10 `src/qonsole/keyevents/qonsole.cpp`

```
[ . . . . ]

bool Qonsole::eventFilter(QObject *o, QEvent *e) {
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *k = static_cast<QKeyEvent*> (e);
        int key = k->key();
        QString str = k->text();
        m_UserInput.append(str);
        updateCursor();
        if ((key == Qt::Key_Return) || (key == Qt::Key_Enter) ) {
            execute();
            return true;           ❶
        }
        else {
            m_Logw->insertPlainText(str);
            return true;
        }
    }
    return QMainWindow::eventFilter(o,e); ❷
}
```

- ❶ We processed the event. This prevents other widgets from seeing it.
 - ❷ Let the base class `eventFilter` have a shot at it.
-

Each time a key is pressed by the user the character generated by that key is appended to the `m_UserInput` string and the position of the cursor in the `LogWindow` is adjusted by the member function `updateCursor()`. When the Enter key is pressed, the member function `execute()` is called so that the command string can be sent to the shell and then reset. Example 12.11 shows the implementation of these two functions.

EXAMPLE 12.11 `src/qonsole/keyevents/qonsole.cpp`

```
[ . . . . ]

bool Qonsole::eventFilter(QObject *o, QEvent *e) {
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *k = static_cast<QKeyEvent*> (e);
        int key = k->key();
        QString str = k->text();
        m_UserInput.append(str);
        updateCursor();
```

continued

```

        if ((key == Qt::Key_Return) || (key == Qt::Key_Enter) ) {
            execute();
            return true;
        }
        else {
            m_Logw->insertPlainText(str);
            return true; ❶
        }
    }
    return QMainWindow::eventFilter(o, e); ❷
}

```

- ❶ We processed the event. This prevents other widgets from seeing it.
- ❷ Let the base class eventFilter have a shot at it.

All that remains to be done is to call the base class function `installEventFilter()` on `m_Logw`, the widget whose events we want to capture. This is done in the constructor, as we see in Example 12.12.

Here is the controller code that sets up `Qonsole`.

EXAMPLE 12.12 `src/qonsole/keyevents/qonsole.cpp`

```

[ . . . . ]

Qonsole::Qonsole() {
    m_Logw = new LogWindow("debug");
    setCentralWidget(m_Logw);
    m_Logw->installEventFilter(this);
    m_Logw->setLineWrapMode(QTextEdit::WidgetWidth);
    m_Bash = new QProcess();
    m_Bash->setReadChannelMode(QProcess::MergedChannels);
    connect (m_Bash, SIGNAL(readyReadStandardOutput()),
            this, SLOT(showOutput()));
    m_Bash->start("bash", QStringList("-i"), QIODevice::ReadWrite);
}

```

EXERCISES: QPROCESS AND PROCESS CONTROL

1. Modify `Qonsole` to support the backspace key.
2. Modify `Qonsole` to support multiple simultaneous terminals in separate tabs.
3. The command `htpasswd`⁴ can be run from the shell, and it provides a command-line interface for encrypting and storing passwords. It stores a list of names and passwords separated by colons in a text file. The passwords are encrypted with a user-selectable protocol.

⁴This command is available on most Linux systems, since it is included with Apache `httpd` server, one of the most widely used Web servers. It is also available on other platforms at no charge.

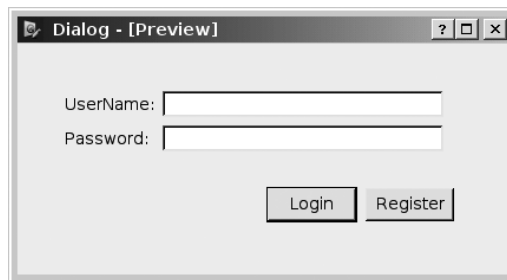
If `htpasswd` has been installed on your *nix system you can learn about its command-line options by viewing its manual page.

Write a Qt wrapper, called `Users`, around the `htpasswd` command. It should have the interface shown in the following figure.

Users
+ containsUser(userName : QString) : bool
+ checkPassword(user : QString, pw : QString) : bool
+ addUser(user : QString, pw : QString) : bool

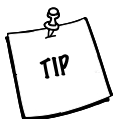
The constructor should load all users and encrypted passwords into a `QMap`. `addUser` will require using `QProcess` to run `htpasswd`. `checkPassword` will use the in-memory map and the `htpasswd -nb` command to verify that the password is correct.

Write a graphical front-end to this application—a login screen, as in the figure that follows.



Use message boxes to describe precisely any errors that occur during login. For example if the user enters the correct username but an incorrect password, the error message should be different from the one for an incorrect username. In a real-world security application, you would not have different error messages (no reason to help would-be hackers gaining access to our systems, right?) but we want a different message to help in testing.

If the user clicks **Register**, show *another* dialog (or modify the existing one) so that it asks for the password twice, to verify that the user entered it correctly.



The authors encountered some difficulties using the `crypt()` (`-d`) and MD5 (`-m`) encryption schemes for this problem, so we recommend using SHA (`-s`) encryption.

12.2 Threads and QThread

Platform-independent threads are a rarity in C++ open-source libraries, because threads are handled in different ways by different operating systems. On the other hand, it is not possible to write GUI applications without threads, since a user click may take just an instant, while the work that needs to be done in response may take much longer. Users are accustomed to being able to continue moving and clicking while their work is being done, and seeing constant progress feedback at the same time.

A single process can simulate multitasking by supporting multiple threads. Each thread has its own call stack and a current statement to execute. A multi-threaded process iterates through the threads, switching to each thread's stack, executing some statements for awhile, and then switching to the next thread. Qt's thread model permits the prioritizing and control of threads.

12.2.1 QPixmap and QThread Animation Example: Movie Player

In this section we write a multithreaded animation viewer application. This application loops through a sequence of eight images on the screen. The user controls the time interval between images to speed up or slow down the process. Figure 12.4 shows the two main classes for this application.

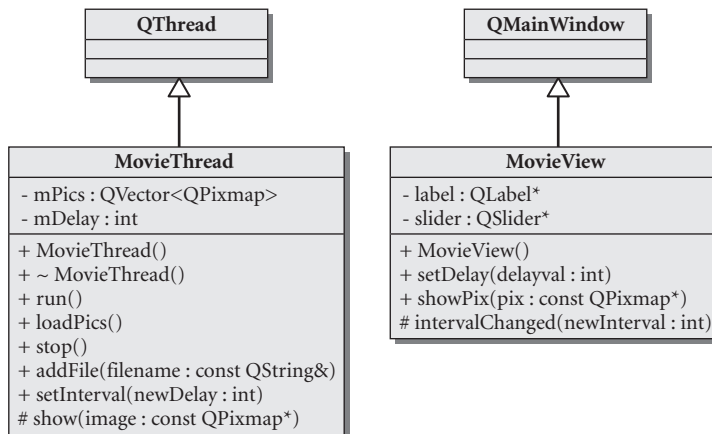


FIGURE 12.4 UML diagram for Movie and MovieView

We will start with the top-level code in Example 12.13 and then drill down to the lower layers of code.

EXAMPLE 12.13 `src/threads/animate/moviethreadmain.cpp`

```
[ . . . . ]
int main(int argc, char** argv) {
    QApplication app(argc, argv);
    MovieView view;
    MovieThread movie;
    app.connect(&movie, SIGNAL(show(const QPixmap*)),
               &view, SLOT(showPix(const QPixmap*)));
    app.connect(&view, SIGNAL(intervalChanged(int)),
               &movie, SLOT(setInterval(int)));
    app.connect(&app, SIGNAL(aboutToQuit()), &movie, SLOT(stop()));
    movie.start(); ❶
    view.show();
    return app.exec();
}

[ . . . . ]
```

- ❶ A new thread starts executing at this point, but the method returns immediately. The new thread starts by calling `movie.run()`.
-

The interface for starting a thread is similar to that of starting a process—in both cases, `start()` returns immediately. Instead of running another program, the newly created thread runs in the same process, shares the same memory, and starts by calling the `run()` function.

The `MovieThread` does not actually display anything—it is a *model* that stores the data representing the movie. Periodically, it emits a signal containing a pointer to the correct pixmap to display as seen in Example 12.14.

EXAMPLE 12.14 `src/threads/animate/moviethread.cpp`

```
[ . . . . ]

void MovieThread::run() {
    int current(0), picCount(m_Pics.size());
    while (true) {
        msleep(m_Delay);
        emit show(&m_Pics[current]);
        current = (current + 1) % picCount;
    }
}
```

In Example 12.15 we can see the slot that actually displays the images and the slot that responds to the slider signals. These slots are in the *view* class, as they should be.

EXAMPLE 12.15 `src/threads/animate/moviewview.cpp`

```
[ . . . . ]

void MovieView::showPix(const QPixmap* pic) {
    label->setPixmap(*pic);
}

void MovieView::setDelay(int newValue) {
    QString str;
    str = QString("%1ms delay between frames").arg(newValue);
    slider->setToolTip(str);
    emit intervalChanged(newValue);
}

```

To summarize: the `run()` function emits a signal from the (model) `MovieThread` object, which will be received by a slot in the (view) `MovieView` object, as arranged by the `connect()` statement in the `main()` function (shown in Example 12.13). Signals are ideal for transmitting data to objects across threads. In Example 12.16 we see how the view is created, and how the slider controls the speed of the animation.

EXAMPLE 12.16 `src/threads/animate/moviewview.cpp`

```
[ . . . . ]

MovieView::MovieView() {
    resize(200, 200);

    slider = new QSlider(Qt::Vertical);
    slider->setRange(1,500);
    slider->setTickInterval(10);
    slider->setValue(100);
    slider->setToolTip("How fast is it spinning?");
    connect(slider, SIGNAL(valueChanged(int)), this,
            SLOT(setDelay(int)));
    QDockWidget *qdw = new QDockWidget("Delay");
    qdw->setWidget(slider);
    addDockWidget(Qt::LeftDockWidgetArea, qdw);
    label = new QLabel("Movie");
    setCentralWidget(label);
}

```

The user controls the time interval between image exposures with a `QSlider` widget that is placed in a dock widget.

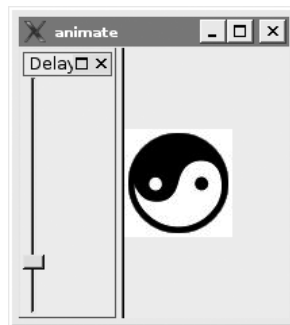
In Example 12.17, to avoid a crash on exit, we introduced a delay that makes sure the thread has enough time to be properly terminated.

EXAMPLE 12.17 `src/threads/animate/moviethread.cpp`

```
[ . . . . ]

void MovieThread::stop() {
    terminate();
    wait(5000);
}
```

Putting it all together, we have produced a movie of a spinning yin-yang symbol, using resources, signals, slots, and threads, with an interface that gives the user control of the spin speed via a docked slider widget. The following figure is a screenshot of the running program.



To load images from disk, we use the Qt resource feature (discussed in Section 11.4), which permits us to “embed” binary files into the executable.

EXAMPLE 12.18 `src/threads/animate/moviethread.cpp`

```
[ . . . . ]

void MovieThread::loadPics() {
    FileVisitor fv("*.jpg");
    connect (&fv, SIGNAL(foundFile(const QString&)),
            this, SLOT(addFile(const QString&)));
    fv.processEntry(":/images/"); ❶
}
```

continued

```
void MovieThread::addFile(const QString& filename) {
    m_Pics << QPixmap(filename);
}
```

- ❶ We are using resources, which link binary files into the executable. They exist in a file system rooted at “.”. See file: `animate.qrc` for list of embedded resources (jpg files).

12.2.2 Movie Player with QTimer

The `MovieThread` object, discussed in Section 12.2.1, simply emitted a signal periodically, with the interval determined by the user. This is a very simple use of a thread.

The `QTimer` class is well suited for emitting periodic signals to drive animations and other rapid but brief operations.⁵ In Example 12.19 we derive our movie model class from `QTimer` instead of deriving it from `QThread`.

EXAMPLE 12.19 `src/threads/animate/movietimer.h`

```
[ . . . . ]
class MovieTimer :public QTimer {
    Q_OBJECT
public:
    MovieTimer();
    ~MovieTimer();
    void loadPics();
public slots:
    void nextFrame();
    void addFile(const QString& filename);
    void setInterval(int newDelay);
signals:
    void show(const QPixmap *image);
private:
    QVector<QPixmap> pics;
    int current;
};
[ . . . . ]
```

The `MovieTimer` class can be used in place of the `MovieThread` class of the previous example. We need to change only one line of the client code from Example 12.13, as we show in Example 12.20.

⁵ Connecting a `QTimer` signal to a slot that takes a relatively long time to execute may slow down the main thread and cause the `QTimer` to miss clock ticks.

EXAMPLE 12.20 src/threads/animate/movietimermain.cpp

```
[ . . . . ]
int main(int argc, char** argv) {
    QApplication app(argc, argv);
    MovieView view;
    MovieTimer movie;
    app.connect(&movie, SIGNAL(show(const QPixmap*)),
               &view, SLOT(showPix(const QPixmap*)));
    app.connect(&view, SIGNAL(intervalChanged(int)),
               &movie, SLOT(setInterval(int)));
    app.connect(&app, SIGNAL(aboutToQuit()), &movie, SLOT(stop()));
    movie.start(); ❶
    view.show();
    return app.exec();
}

[ . . . . ]
```

❶ Starts the timer, not a thread.

In fact, the implementation of `MovieTimer` is simpler than that of `MovieThread`, as shown in Example 12.21.

EXAMPLE 12.21 src/threads/animate/movietimer.cpp

```
[ . . . . ]

MovieTimer::MovieTimer(): current(0) {
    setInterval(100);
    loadPics();
    connect(this, SIGNAL(timeout()), this, SLOT(nextFrame()));
}

void MovieTimer::nextFrame() {
    current = (current + 1) % pics.size();
    emit show(&pics[current]);
}

[ . . . . ]
```

`QTimer` is very convenient and can be used in some situations where, at first, you might think of using a `QThread`. A `QTimer` with a timeout of 0 will emit its signal as fast as possible, but only when the Qt event queue is empty. By rewriting a “worker thread” to operate in small chunks when its slot is repeatedly called, you can achieve something similar to multithreading without slowing down the user interface response.

12.2.3 Multiple Threads, Queues, and Loggers

Example: Giant

In this section, we present an example of a **consumer-producer** problem inspired by the famous fable, “Jack and the Beanstalk.” As we see in Figure 12.5, the main window of the application shows a split-screen view (provided by the `QSplitter`) of two text areas.

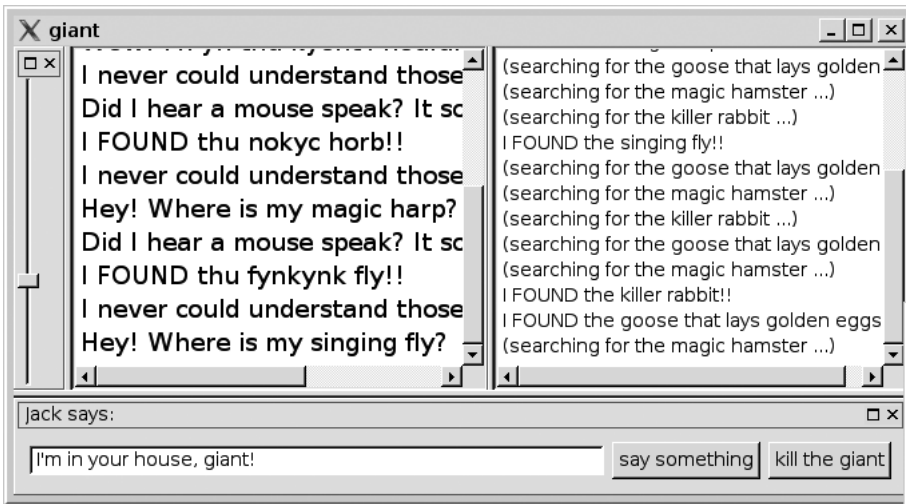


FIGURE 12.5 Giant versus Jack main window

The slider docked on the left determines the rate at which Jack steals the items on his list from the Giant. That rate has a random component. The `LineEdit` docked on the bottom allows the user to give Jack things to say to the Giant. Each text area is a `LogWindow` that monitors a named `Logger`. Both classes are provided in `libutils` and are documented in our API Docs.⁶

A `Logger` is an abstraction for a named stream of bytes. A `Logger` gets attached to a `LogWindow` through signals and slots.

Loggers are important tools for debugging (and, sometimes, even visualizing) multitasking processes. These are based on the Java loggers, but take advantage of

⁶ We discussed how to obtain and install `libutils` in Section 7.4.

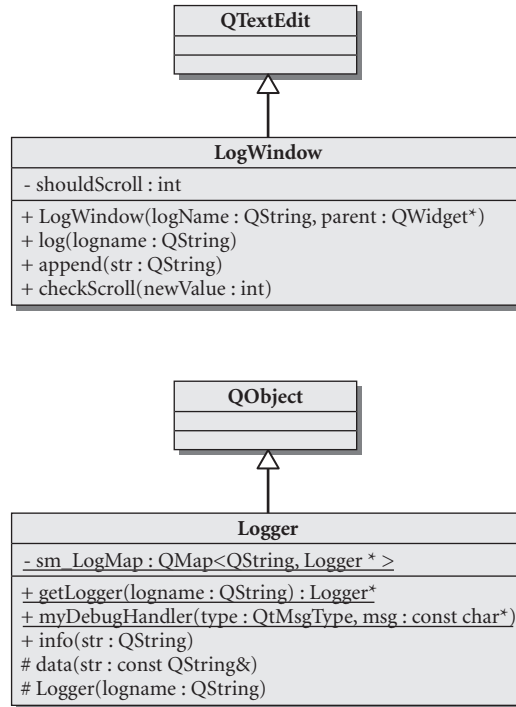


FIGURE 12.6 Loggers

Qt's signals and slots, which can send messages and objects across threads. The UML diagram in Figure 12.6 shows how these classes are related.

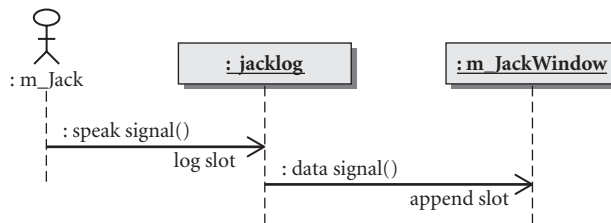


FIGURE 12.7 Speaking with signals

When this program executes, two threads are running, Jack and Giant. Each thread emits signals and writes to loggers independently. The sequence diagram in Figure 12.7 shows how this works in one thread.

Jack and Giant have no knowledge of each other—they are strictly decoupled. They do not include each other's header files, and they are connected to each

other exclusively by signals and slots in controller code from `GiantWindow`. These relationships are depicted in the UML diagram in Figure 12.8.

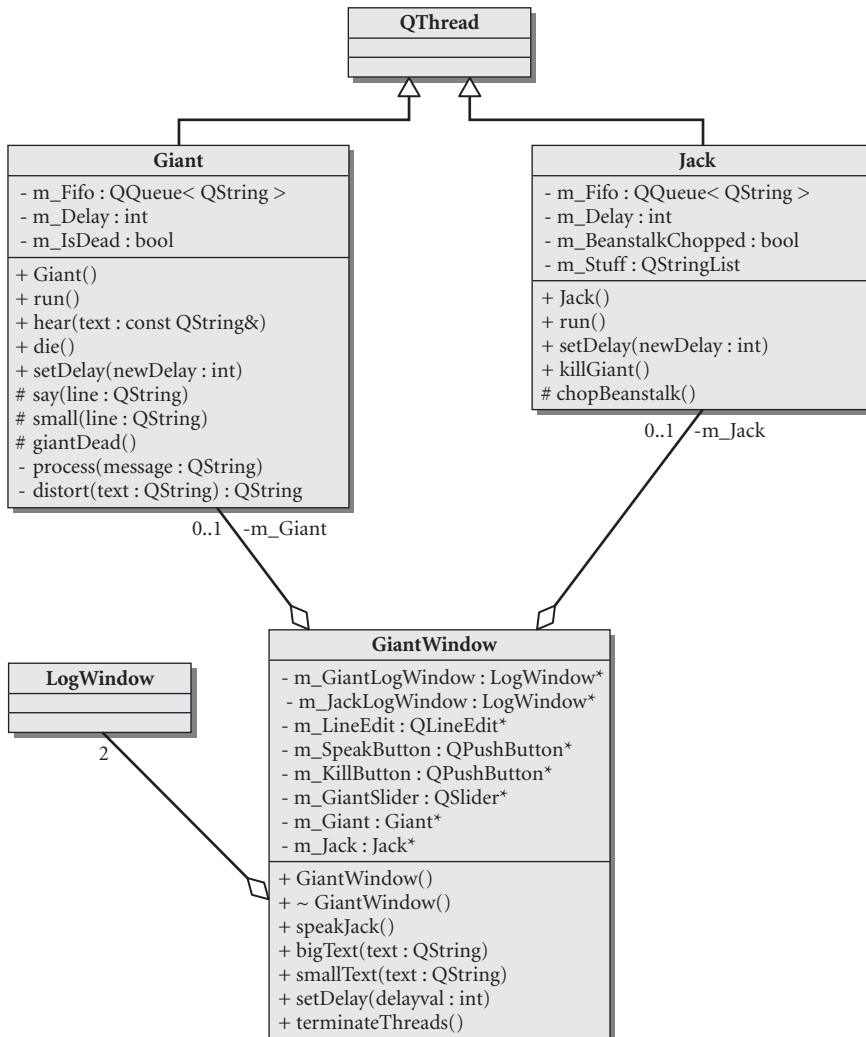


FIGURE 12.8 Jack and Giant UML

Each thread has an incoming message queue (fifo). Messages may arrive in the Giant's queue faster than he can handle them. Some messages will be responded to, some will be ignored.

To make things more interesting, we have connected Jack's `Logger`'s data signal to the Giant's `hear` slot, so the Giant is somewhat aware of what Jack is doing and can react to it. We show the class definition for `Giant` in Example 12.22.

EXAMPLE 12.22 src/threads/giant/giant.h

```
[ . . . . ]
class Giant :public QThread {
    Q_OBJECT
public:
    Giant();
    void run();
public slots:
    void hear(const QString& text);
    void die();
    void setDelay(int newDelay);
signals:
    void say(QString line);
    void small(QString line);
    void giantDead();
private:
    void process(QString message);
    QString distort(QString text);
    Queue<QString> m_Fifo;
    int m_Delay;
    bool m_IsDead;
};
[ . . . . ]
```

Even without threads, both Jack and Giant have slots and receive signals emitted from the Qt event loop, as we see in Example 12.23.

EXAMPLE 12.23 src/threads/giant/giant.cpp

```
[ . . . . ]

void Giant::die() {
    if (m_IsDead)
        return;
    m_IsDead = true;
    m_Fifo << "What? You nasty little worm.";
    m_Fifo << "I will squash you!!! ";
    m_Fifo << "So you're running back down the beanstalk.";
    m_Fifo << "I am coming right after you!";
    msleep(m_Delay); ❶
    m_Fifo << "Oh no!! Someone chopped the beanstalk!!";
    m_Fifo << "aaaaaaaaa!!! .....";
    m_Fifo << " *splat* \n";
}

void Giant::hear(const QString &text) {
    QString t2 = ":" + text;
    m_Fifo << t2;
}

}
```

❶ Which thread is going to sleep here, the Giant or the MainThread?

In addition to responding to signals from the main thread, Jack and Giant each override the `run()` function. Example 12.24 shows the Giant's thread code, which is a typical "infinite" loop, reading input when there is some and sleeping when there is not.

To simulate the Giant's difficulty hearing the sounds made by a creature as tiny as Jack, we use the `distort()` function to introduce some noise into his `process()` function.

EXAMPLE 12.24 `src/threads/giant/giant.cpp`

```
[ . . . . ]

void Giant::run() {
    int zcount = 0;
    while (true) {
        zcount = 0;
        while (m_Fifo.isEmpty()) {
            msleep(m_Delay);
            ++zcount ;
            if (m_IsDead) {
                emit giantDead();
                break;
            }
            if (zcount > 3) {
                m_Fifo << "zzzzzz";
            }
        }
        QString message = m_Fifo.dequeue();
        msleep(m_Delay);
        process(message);
    }
}

void Giant::process(QString message) {
    if(message.startsWith(":")) {
        QStringList l = message.split(":");
        msleep(m_Delay);
        if(! l[1].startsWith("(")) {
            QString msg = l[1];
            emit say ("Did I hear a mouse speak? It sounded like");
            emit say (distort(msg));
            emit say ("I never could understand those darned mice.");
            if (msg.startsWith("I FOUND")) {
                msg = msg.remove("I FOUND the ").remove("!!");
                msg = QString("Hey! Where is my %1?").arg(msg);
                emit say(msg);
            }
        }
    } else
        emit say(message);
}

```

GiantWindow is a very primitive GUI front end for Giant. It has slots that can be hooked up to the giant's signals, as we see in Example 12.25.

EXAMPLE 12.25 `src/threads/giant/giantwindow.h`

```
[ . . . . ]
class GiantWindow : public QMainWindow {
    Q_OBJECT
public:
    GiantWindow();
    ~GiantWindow();
public slots:
    void speakJack();
    void bigText(QString text);
    void smallText(QString text);
    void setDelay(int delayval);
    void terminateThreads();
private:
    LogWindow *m_GiantLogWindow;
    LogWindow *m_JackLogWindow;
    QLineEdit *m_LineEdit;
    QPushButton *m_SpeakButton, *m_KillButton;
    QSlider *m_GiantSlider;
    Giant* m_Giant;
    Jack* m_Jack;
};
[ . . . . ]
```

The constructor spells out the details of arranging things on the screen and connecting signals with slots. Example 12.26 shows some of those details.

EXAMPLE 12.26 `src/threads/giant/giantwindow.cpp`

```
[ . . . . ]
GiantWindow::GiantWindow() {
    resize(800, 600);
    m_Giant = new Giant();
    m_Jack = new Jack();
    /* The giant talks to the GiantWindow through signals and slots */
    connect (m_Giant, SIGNAL(say(QString)), this,
            SLOT(bigText(QString)));
    connect (m_Giant, SIGNAL(small(QString)), this,
            SLOT(smallText(QString)));
    connect (m_Jack, SIGNAL(chopBeanstalk()), m_Giant, SLOT(die()));
    connect (m_Giant, SIGNAL(giantDead()), this,
            SLOT(terminateThreads()));
}
```

continued

```

m_GiantLogWindow = new LogWindow("giant");
m_GiantLogWindow->setToolTip("This is what the giant says");

m_JackLogWindow = new LogWindow("jack");           ❶
m_JackLogWindow->setToolTip("This is what Jack is doing");
Logger *jackLog = Logger::getLogger("jack");

connect (jackLog, SIGNAL(data(const QString&)),
         m_Giant, SLOT(hear(const QString&)));      ❷
QSplitter *split = new QSplitter(this);           ❸
split->addWidget(m_GiantLogWindow);
split->addWidget(m_JackLogWindow);                 ❹
setCentralWidget(split);

[ . . . . ]

```

- ❶ A `LogWindow` will display all messages sent to the logger of the same name. In this case, `Logger::getLogger("jack")` will return an object through which you send messages that get displayed in this window.
- ❷ The giant can hear what Jack is saying!!
- ❸ split-window container with draggable splitter between two widgets
- ❹ We can only add two widgets to split.

12.2.4 Thread Safety and QObjects

A `QObject` that was created in a particular thread “belongs” to that thread and its children must also belong to the same thread. Having parent-child relationships that cross over threads is forbidden by Qt.

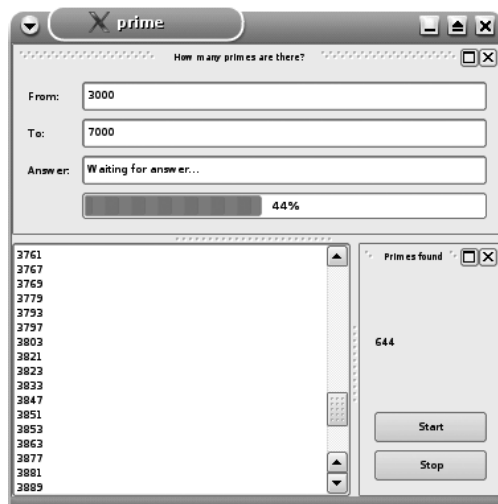
A **thread-safe** object is one that can be accessed concurrently by multiple threads and is guaranteed to always be in a “valid” state. `QObject`s are not “thread safe” by default. To make an object thread safe, there are a number of approaches to take. Some are listed here, but we recommend the Qt 4 Thread Support⁷ documentation for further details.

1. `QMutex`, for mutual exclusion, along with `QMutexLocker`, allows an individual thread `T` to protect (lock) an object or a section of code so that no other thread can access it until `T` releases (unlocks) it.
2. `QWaitCondition`, combined with `QMutex`, can be used to put a thread into a non-busy block state where it can wait for another thread to wake it up.
3. A `QSemaphore` is a more generalized `QMutex` for situations where a thread may need to lock more than one resource before doing its work. Semaphores make it possible to ensure that a thread only locks resources when enough are available for it to do its job.

⁷ <http://oop.mcs.suffolk.edu/qtdocs/threads.html>

EXERCISE: THREADS AND QTHREAD

- Write a program that tells the user how many prime numbers exist in a range between two numbers supplied in the command line.
 - Make the progress bar show the progress of the search (estimated number of numbers left to check, relative to number of numbers checked). Also show how many primes have been found so far.
 - Make the start/stop buttons start/stop the search for prime numbers. Make sure the search thread does not take up so much CPU time that you are unable to perform other user interactions.
- Create a `QListWidget` (or `QListView`) that shows the prime numbers found so far during the search.
- Display the elapsed search time and the number of primes found per second.



12.3 Summary: QProcess and QThread

`QProcess` and `QThread` are similar in concept and have one special thing in common: a powerful `start()` function that causes an execution *fork* (two

“things” are happening, when before there was only one). Table 12.1 summarizes the main differences between a process and a thread.

TABLE 12.1 QProcess versus QThread

QProcess	QThread
Runs a (presumably) different program in separate process, separate memory.	Runs in the same process, shares code, and shares memory with other threads.
Communicates with child process via streams (<code>stdin</code> , <code>stdout</code> , <code>stderr</code>) and passes in data initially using command line arguments (<code>argc</code> , <code>argv</code>) or environment variables (<code>getenv()</code>). See Appendix B.	Shares memory and code with peer threads. Synchronizes using locks, wait conditions, mutexes, and semaphores.
Managed by the operating system	Managed by the process

REVIEW QUESTIONS

1. What are two important differences between a process and a thread?
2. List and explain at least two mechanisms by which a parent process communicates information to its child process.
3. List and explain at least two mechanisms by which threads synchronize with each other.
4. In what situations can a `QTimer` be used instead of a `QThread`? Why would one want to do that?

13

CHAPTER 13

Validation and Regular Expressions

Validation of input data is an important issue that can be handled in an object-oriented way by a combination of Qt classes. This chapter discusses some efficient ways of validating input, including the use of regular expressions.

13.1 Validators	308
13.2 Regular Expressions	310
13.3 Regular Expression Validation	316

13.1 Validators

Validators are nonvisual objects that are attached to input widgets (such as `QLineEdit`, `QSpinBox`, and `QComboBox`) to provide a general framework for checking user input. Qt has an abstract class named `QValidator` that establishes the interface for all built-in and custom validators.

There are two concrete subclasses that can be used for numeric range checking: `QIntValidator` and `QDoubleValidator`. There is also a concrete subclass that can be used for validating a string with a specified regular expression. We will discuss regular expressions in the next section.

`QValidator::validate()` is a pure virtual method that returns an enumerated value, as follows:

- **Invalid:** The expression does not satisfy the required conditions and further input will not help.
- **Intermediate:** The expression does not satisfy the required conditions, but further input might produce an acceptable result.
- **Acceptable:** The expression satisfies the required conditions.

Other member functions enable the setting of the conditions that `validate()` uses.

In Example 13.1 we have a short app that uses the two numerical validators. It takes an `int` and a `double` from the user to display the product. Each input is given a range check when the user presses return.

EXAMPLE 13.1 `src/validate/inputform.h`

```
[ . . . . ]
class InputForm : public QMainWindow {
    Q_OBJECT
public:
    InputForm(int ibot, int itop, double dbot, double dtop);
    void setupForm();
public slots:
    void computeResult();
```



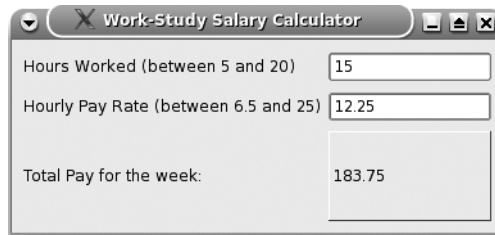
```
private:
    int m_BotI, m_TopI;
    double m_BotD, m_TopD;
    QIntValidator* m_IValid;
    QDoubleValidator* m_DValid;
    QGridLayout* m_Layout;
    QLineEdit *m_IntEntry, *m_DoubleEntry;
    QLabel* m_Result;
    QWidget* m_Center;
};
[ . . . . ]
```

In Example 13.2, validators are initialized with range values in the constructor and are assigned to their respective input widgets in the `setupForm()` function.

EXAMPLE 13.2 `src/validate/inputform.cpp`

```
[ . . . . ]
InputForm::InputForm(int ibot, int itop, double dbot, double dtop):
    m_BotI(ibot), m_TopI(itop), m_BotD(dbot), m_TopD(dtop),
    m_IValid(new QIntValidator(ibot, itop, this)),
    m_DValid(new QDoubleValidator(dbot, dtop, 2, this)),
    [ . . . . ]
void InputForm::setupForm() {
    [ . . . . ]
    m_IntEntry->setValidator(m_IValid);
    m_DoubleEntry->setValidator(m_DValid);
    connect(m_IntEntry, SIGNAL(returnPressed()),
            this, SLOT(computeResult()));
    connect(m_DoubleEntry, SIGNAL(returnPressed()),
            this, SLOT(computeResult()));
}
[ . . . . ]
```

The running program looks like the screenshot that follows.



If you run it, try to enter a non-`int` textfield, or a non-`float` in the second. You should notice that it does not recognize invalid characters as input.

13.2 Regular Expressions

Regular expressions are powerful tools for validating input, for extracting data from input, and for searching and replacing (see Table 13.1). The regular expression pattern matching language is used to describe regular expressions. A regular expression (*regex* for short) describes constraints on the way a string is composed.

TABLE 13.1 Examples of Regular Expressions

Pattern	Meaning
hello	matches the literal string, <code>hello</code>
<code>c*at</code>	(quantifier) zero or more occurrences of <code>c</code> , followed by <code>at</code>
<code>c?at</code>	zero or 1 occurrences of <code>c</code> , followed by <code>at</code>
<code>c.t</code>	(character set: <code>anychar</code>) <code>c</code> followed by any character followed by <code>t</code> : <code>cat</code> , <code>cot</code> , <code>c3t</code> , <code>ct</code>
<code>c.*t</code>	(char set and quantifier) <code>c</code> followed by 0 or more <code>anychar</code> followed by <code>t</code> : <code>ct</code> , <code>caaatttt</code> , <code>carsdfsdf</code>
<code>ca+t</code>	(quantifier) <code>+</code> means 1 or more of the preceding “thing,” so this matches <code>cat</code> , <code>caat</code> , <code>caaat</code> , but not <code>ct</code>
<code>c\\. *t</code>	(literals) backslashes precede special characters to “escape them,” so this matches the literal: <code>c. *t</code>
<code>c\\.\\. .t</code>	(literals) <code>c\\. .t</code>
<code>c[0-9a-c]+z</code>	matches <code>c312abbaz</code> <code>caa211bac2z</code>
<code>the (cat dog) ate</code> <code>the (fish mouse)</code>	(alternation) <code>the cat ate the fish</code> or <code>the dog ate the mouse</code> or <code>the dog ate the fish</code> , and the obvious last one.
<code>\\w+</code>	(charset) a sequence of alphanumerics, or a word, same as <code>[a-zA-Z0-9]+</code>
<code>\\W</code>	a character that is not part of a word (punctuation, whitespace, etc.)
<code>\\s{5}</code>	exactly 5 whitespace chars (tabs, spaces, newlines)
<code>\\S{1,5}</code>	at least 1, at most 5 non-whitespace (printable characters)
<code>\\d</code>	a digit <code>[0-9]</code> (and <code>\\D</code> is a non-digit, i.e., <code>[^0-9]</code>)
<code>\\d{3}-\\d{4}</code>	7-digit phone numbers: <code>555-1234</code>
<code>\\b[A-Z]\\w+</code>	<code>\\b</code> means word boundary: matches <code>mBuffer</code> but not <code>StreamBuffer</code>

Regular expressions were first available in tools such as `vi`, `emacs`, `awk`, `sed`, and the POSIX Standard Library. Perl was the first mainstream programming language to integrate regular expressions so tightly into the language that it caused many people to learn regular expressions for the first time. Many enhancements have been made to the version of regular expressions that Perl recognizes. The enhancements are part of what we call Perl-style *extended* regular expressions. These extended regular expressions are also available in Java and Python.

Qt has a class, `QRegExp`, that implements most of the Perl-style extended regular expression language.

13.2.1 Regular Expression Syntax

A regular expression can be a simple string, in which case it specifies an exact string match, or it can be a string that includes regular expression **meta-characters**. A meta-character is a character that describes *other* characters.

Here are some of the most commonly used meta-characters.

1. Special characters
 - `.` (the dot matches any character)
 - `\n` (matches the newline character)
 - `\f` (matches the form feed character)
 - `\t` (matches the tab character)
 - `\xhhhh` (matches the Unicode character whose code is the hexadecimal number `hhhh` in the range `0x0000` to `0xFFFF`)
2. Quantifiers: Modifiers that specify the number of occurrences of the preceding character (or group) that may appear in the matching expression.
 - `+` (1 or more occurrences)
 - `?` (0 or 1 occurrences)
 - `*` (0 or more occurrences)
 - `{i, j}` (at least `i` but not more than `j` occurrences)
3. Character sets: Sets of allowable values for the character in the specified position of the matching expression. Several character sets are predefined:
 - `\s` (matches any whitespace character)
 - `\S` (matches any non-whitespace character)
 - `\d` (matches any digit character: `'0'` to `'9'`)
 - `\D` (matches any non-digit character)

- `\w` (matches any “word” character; i.e., any letter or digit or the underscore ‘_’)
- `\W` (matches any non-word character)

Character sets can also be specified in square brackets:

- `[AEIOU]` (matches any of the chars ‘A’, ‘E’, ‘I’, ‘O’, or ‘U’)
 - `[a-g]` (the dash makes this a range from ‘a’ to ‘g’)
 - `[^xyz]` (matches any character *except* for ‘x’, ‘y’, and ‘z’)
4. Grouping and capturing characters (round parentheses): Characters that can be used to form a group. Groups can be back-referenced, meaning that if there is a match, the grouped values can be captured and accessed in various ways.

For convenience, up to 9 groups can be referenced within the regular expression by using the identifiers `\1` thru `\9`.

There is also a `QRegExp` member function `cap(int nth)` that returns the *n*th group (as a `QString`).

5. Anchoring characters: Assertions that specify the boundaries of a matching effort.
- The caret (`^`), if it is the first character in the regex, indicates that the match starts at the beginning of the string.
 - The dollar sign (`$`), when it is the last character in the regex, means that the effort to match must continue to the end of the string.
 - In addition, there are word boundary (`\b`) or non-word boundary (`\B`) assertions that help to focus the attention of the regex.



Backslashes are used for escaping special characters in C++ strings as well, so this means that regular expression strings inside C++ strings must be “double-backslashed.” In other words, every `\` becomes `\\`, and to match the backslash character itself you will need four: `\\\\`

There is much more to regular expressions. Time spent learning to use them is well-invested time. The documentation for `QRegExp` is a good place to start. For a much more extensive discussion, we recommend [Fried198].

13.2.2 Regular Expressions: Phone Number Recognition

The Problem

We want to specify conditions, in a generic way, that must be satisfied by input data at runtime. For example:

- In a U.S. address, every zip code can have five digits, followed by an optional dash (-) and four more digits.
- A U.S. phone number consists of ten digits, usually grouped 3 + 3 + 4, with optional parentheses and dashes and an optional initial 1.
- A U.S. state abbreviation must be one from the set of 50 approved abbreviations.

How can we impose conditions such as these on incoming data in an object-oriented way?

Suppose that you wanted to write a program that recognized phone number formats, and could accept Dutch or U.S./Canada phone numbers. You would need to take the following things into consideration.

- For any U.S./Canada format numbers, there must be AAA EEE NNNN, where A = area code, E = exchange, and N = number.
- For Dutch format numbers, there must be CC MM followed by either NN NN NNN or NNN NNNN, where C = country code, M = municipal code, and N = localNumberDigits.
- There might be dashes or spaces between number clusters.
- There might be + or 00 in front of the country code.

Imagine how you would write this program using the standard tools available to you in C++. It would be necessary to write lengthy parsing routines for each possible format. Example 13.3 shows the desired output of such a program.

EXAMPLE 13.3 src/regexp/testphone.txt

```
$> ./testphone
Enter a phone Number (or q to quit): 1112223333
validated: (US/Canada) +1 111-222-3333
Enter a phone Number (or q to quit): 20618676017
validated: (Europe) +20 (0)61-86-76-017
Enter a phone Number (or q to quit): 31206472582
validated: (Europe) +31 (0)20-64-72-582
Enter a phone Number (or q to quit): 16175551212
validated: (US/Canada) +1 617-555-1212
```

continued

```

Enter a phone Number (or q to quit): +31 20 64 28 258
validated: (Europe) +31 (0)20-64-28-258
Enter a phone Number (or q to quit): 1 (617) 222 3333
validated: (US/Canada) +1 617-222-3333
Enter a phone Number (or q to quit): 31 20 111 1111
validated: (Europe) +31 (0)20-11-11-111
Enter a phone Number (or q to quit): asdf
Unknown format
Enter a phone Number (or q to quit): 111 2222 333
Unknown format
Enter a phone Number (or q to quit): 111223
Unknown format
Enter a phone Number (or q to quit): 1112222333
validated: (US/Canada) +1 111-222-2333
Enter a phone Number (or q to quit): q
$ >

```

A procedural C-style solution that shows how to use `QRegExp` is shown in Example 13.4.

EXAMPLE 13.4 `src/regexp/testphoneread.cpp`

```

[ . . . . ]
QRegExp usformat
("(\++?1[- ]?)?\s+(\d{3})\s)?[\s-]?(\d{3})[\s-]?(\d{4})"); ❶

QRegExp nlformat
("(\+|00)?[\s\-\-]?(\d{3})[\s\-\-]?(\d{3})[\s\-\-]?(\d{3})"); ❷

QRegExp nlformat2
("(\d{3})(\d{3})(\d{3})"); ❸

QRegExp filtercharacters
("[\s-\+\(\)\-\-]"); ❹

QString stdinReadPhone() { ❺
    QString str;
    bool format=false;
    do { ❻
        cout << "Enter a phone Number (or q to quit): ";
        cout.flush();
        str = cin.readLine();
        if (str=="q")
            return str;
        if (usformat.exactMatch(str)) {
            format = true;
            QString areacode = usformat.cap(2);
            QString exchange = usformat.cap(3);
            QString number = usformat.cap(4);
            str = QString(" (US/Canada) +1 %1-%2-%3")
                .arg(areacode).arg(exchange).arg(number);
        }
    }
}
[ . . . . ]

```

```

if (format == false) {
    cout << "Unknown format" << endl;
}
} while (format == false) ;
return str;
}

int main() {
    QString str;
    do {
        str = stdinReadPhone();
        if (str != "q")
            cout << "validated: " << str << endl;
    } while (str != "q");
    return 0;
}
[ . . . . ]

```

- ❶ All usformat numbers have country code 1, and have $3 + 3 + 4 = 10$ digits. White spaces, dashes, and parentheses between these digit groups are ignored, but they help to make the digit groups recognizable.
- ❷ Netherlands (country code 31) numbers have $2 + 2 + 7 = 11$ digits.
- ❸ The last seven digits will be arranged as $2 + 2 + 3$.
- ❹ These are characters we ignore in the last seven digits of NL numbers.
- ❺ Ensures that the user-entered phone string complies with a regular expression and extracts the proper components from it.
- ❻ Keep asking until you get a valid number.

EXERCISES: REGULAR EXPRESSIONS: PHONE NUMBER RECOGNITION

1. Rewrite the birthday reminder application from Section 3.6 so that it accepts dates in any of the following formats:
 - YYYY-MM-DD
 - YYYY/MM/DD
 - MM/DD (year defaults to this year)
 - MMM DD YYYY (year optional, defaults to this year) (MMM = 3-letter abbreviation for month, ignores upper/lower case but does verify validity of month abbreviation)
2. Many operating systems come with a reasonably good list of words that are used by various programs to check spelling. On *nix systems the word list is generally named (at least indirectly) "words". You can locate the file on your *nix system by typing the command

```
locate words | grep dict
```

Piping the output of this command through `grep` reduces the output to those lines that contain the string "dict".

Once you have located your system word list file, write a program that will read lines from the file and, using a suitable regex, display all the words that do the following:

- a. Begin with a pair of repeated letters
- b. End in “gory”
- c. Have more than one pair of repeated letters
- d. Are palindromes
- e. Consist of letters arranged in strictly increasing alphabetic order (e.g., knot)

If you cannot find such a suitable word list on your system, you can use the file `canadian-english-small.gz`, in our dist directory.¹ After you download it, you must uncompress it with the command

```
gunzip canadian-english-small.gz
```

13.3 Regular Expression Validation

The class `QRegExpValidator` uses a `QRegExp` to validate an input string. In Example 13.5, we defined a main window class that contains a `QRegExpValidator` and some input widgets.

EXAMPLE 13.5 `src/validate/regexval/rinputform.h`

```
[ . . . . ]
class RinputForm : public QMainWindow {
    Q_OBJECT
public:
    RinputForm();
    void setupForm();
public slots:
    void computeResult();
private:
    QRegExpValidator* m_PhoneValid;
    QGridLayout* m_Layout;
    QLineEdit* m_PhoneEntry;
    QWidget* m_Center;
    QLabel* m_PhoneResult;
    QString m_Phone;
    static QRegExp sm_PhoneFormat;
};
[ . . . . ]
```

¹ <http://oop.mcs.suffolk.edu/dist>

We borrowed the regex from Example 13.4 and used it to initialize the static `QRegExp`. The program in Example 13.6 takes a phone number from the user and displays it only if it is valid.

EXAMPLE 13.6 `src/validate/regexval/rinputform.cpp`

```
[ . . . . ]
QRegExp RinputForm::sm_PhoneFormat (
    "(\\+?1[- ]?)?\\((?\\d{3,3})\\)?[\\s-]?\\d{3,3}[\\s-]?\\d{4,4})";

RinputForm::RinputForm() :
    m_PhoneValid(new QRegExpValidator(sm_PhoneFormat, this)),
    [ . . . . ]
void RinputForm::setupForm() {
    [ . . . . ]
    m_PhoneEntry->setValidator(m_PhoneValid);
    connect(m_PhoneEntry, SIGNAL(returnPressed()),
           this, SLOT(computeResult()));
}

void RinputForm::computeResult() {
    m_Phone = m_PhoneEntry->text();
    if (sm_PhoneFormat.exactMatch(m_Phone)) {
        QString areacode = sm_PhoneFormat.cap(2);
        QString exchange = sm_PhoneFormat.cap(3);
        QString number = sm_PhoneFormat.cap(4);
        m_PhoneResult->setText(QString("(US/Canada) +1 %1-%2-%3")
                               .arg(areacode).arg(exchange).arg(number));
    }
}

[ . . . . ]
```

The `QRegExpValidator` will not permit entry of any characters that would produce an invalid result. Here is a snapshot of the running program.



As we have seen, the `QValidator` classes provide a powerful mechanism for validating input data. Qt has provided two numerical range validators and a regular expression validator. If other types of data need to be validated, it is not difficult to produce other subclasses of `QValidator`.

EXERCISES: VALIDATION AND REGULAR EXPRESSIONS

1. Write a program that extracts the hyperlinks from HTML files using regular expressions. Each hyperlink looks like this:

```
<a href="http://www.web.www/location/page.html">The Label</a>
```

For each hyperlink encountered in the input file, print just the URL and label, separated by a tab.

Keep in mind that optional whitespace can be found in different parts of the above example pattern. Test your program on a variety of different Web pages and verify that your program does indeed catch all the links.

2. You just changed companies and you want to reuse some open-source code that you wrote for the previous company. Now you need to rename all the data members in your source code. The previous company wanted data members named like this: `mVarName`, but your new company wants them named like this: `m_varName`. Extend `FileVisitor` and perform a substitution in the text of each visited file so that all data members conform to the new company's coding standards.
3. Write a program that asks the user for a palindrome in a `QLineEdit` and has a validator that enforces the rule that the letters (ignoring uppercase and lowercase and whitespace) do indeed have the same order in either direction.

REVIEW QUESTIONS

1. What is a regular expression? What can you use it for?
2. What is a validator? What is it used for?
3. What is a regular expression meta-character? There are four kinds of metacharacters: quantifier, character set, group, and anchor. Give examples of each type, and explain what they mean.


14

CHAPTER 14

Parsing XML

This chapter introduces two ways of parsing XML data, available from Qt's XML module. We demonstrate event-driven parsing with SAX, the Simple API for XML, and tree-style parsing with DOM, the Document Object Model.

14.1	The Qt XML Module	325
14.2	Event-Driven Parsing	325
14.3	XML, Tree Structures, and DOM	329



XML is an acronym for eXtensible Markup Language. It is a markup language similar to HTML (HyperText Markup Language), but with stricter syntax and no semantics (i.e., no meanings associated with the tags).

XML's stricter syntax is in strong contrast to HTML. For example:

- Each XML `<tag>` must have a closing `</tag>`, or be self-closing, like this: `
`.
- XML tags are case sensitive: `<tag>` is not the same as `<Tag>`.
- Characters such as `>` and `<` that are not actually part of a tag must be replaced by passive equivalents such as `>`; and `<`; in an XML document to avoid confusing the parser.

Example 14.1 is an HTML document that does not conform to XML rules.

EXAMPLE 14.1 `src/xml/html/testhtml.html`

```
<html>
<head> <title> This is a title </title>

  <!-- Unterminated <link> and <input> tags are quite commonplace
        in HTML code:      -->
  <link rel="Saved Searches" title="oopdocbook"
        href="buglist.cgi?cmdtype=runnamed&namedcmd=oopdocbook">
  <link rel="Saved Searches" title="queryj"
        href="buglist.cgi?cmdtype=runnamed&namedcmd=queryj">
</head>
<body>
<p> This is a paragraph. What do you think of that? </p>

Html makes use of unterminated line-breaks: <br>
And those do not make XML parsers happy. <br>

<ul>
<li> HTML is not very strict.
<li> An unclosed tag doesn't bother HTML parsers one bit.
</ul>

</body>
</html>
```

If we combined XML syntax with HTML element semantics, we would get a language called XHTML. Example 14.2 shows Example 14.1 rewritten as XHTML.

EXAMPLE 14.2 `src/xml/html/testxhtml.html`

```
<!DOCTYPE xhtml >
<html>
<head>
<title> This is a title </title>
<!-- These links are now self-terminated: -->
<link rel="Saved&nbsp;Searches" title="oopdocbook"
      href="buglist.cgi?cmdtype=runnamed" />
<link rel="Saved&nbsp;Searches" title="queryj"
      href="buglist.cgi?namedcmd=queryj" />
</head>
<body>

<p> This is a paragraph. What do you think of that? </p>

<p>
Html self-terminating linebreaks are ok: <br/>
They don't confuse the XML parser. <br/>
</p>

<ul>
<li> This is proper list item </li>
<li> This is another list item </li>
</ul>

</body>
</html>
```

XML is a whole class of file formats that is understandable and editable by humans as well as by programs. XML has become a popular format for storing and exchanging data from Web applications. It is also a natural language for representing hierarchical (tree-like) information, which includes most documentation.

Many applications (e.g., Qt Designer, Umbrello, Dia) use an XML file format for storing data. Qt Designer's .ui files use XML to describe the layout of Qt widgets in a GUI. The book you are reading now is written in a flavor of XML called Slacker's DocBook.¹ It's like DocBook,² an XML language for writing books, but it adds some shorthand tags from XHTML and custom tags for describing courseware.

¹ <http://slackerdoc.tigris.org/>

² <http://www.docbook.org>

An XML document is comprised of **nodes**. Elements *are* nodes and look like this: `<tag> text or elements </tag>`. An opening tag can contain **attributes**. An attribute has the form: `name="value"`. Elements nested inside one another form a parent-child tree structure.

EXAMPLE 14.3 `src/xml/sax1/samplefile.xml`

```
<section id="xmlintro">
  <title> Intro to XML </title>
  <para> This is a paragraph </para>
  <ul>
    <li> This is an unordered list item. </li>
    <li c="textbook"> This only shows up in the textbook </li>
  </ul>
  <p> Look at this example code below: </p>
  <include src="xmlsamplecode.cpp" mode="cpp"/>
</section>
```

In Example 14.3, `` has two `` children, and its parent is a `<section>`. Elements with no children can be self-terminated with a `/>`, i.e., `<include/>`. Some elements such as `<section>` and `<include>` have attributes. Indenting nested elements helps readability, but extra whitespace is ignored by most parsers.



How many direct children are there of the `<section>`?

XML Editors

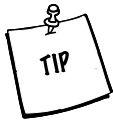
There are several open-source XML editors available. You are encouraged to try them before you go with a commercial solution.

1. jEdit³ has an XML plugin that works quite well. Be sure to use this option: "insert closing tag when `</` is typed".
2. For KDE users, there is *quanta*.⁴ Like *kdevelop*, this is based on *Kate*, the KDE advanced text editor. If you are accustomed to using emacs keys, be sure to get this *Kate* plugin: *ktexteditor-emacsextensions*.⁵

³ <http://www.jedit.org>

⁴ <http://quanta.kdewebdev.org/>

⁵ <http://www.kde-apps.org/content/show.php?content=21706>



XMLLINT The free tool `xmllint` is very handy for checking an XML file for errors. It reports very descriptive error messages (mismatched start/end tags, missing characters, etc.) and points out where the errors are. It can also be used to indent/“pretty print” a well-formed XML document.

14.1 The Qt XML Module

Qt’s XML Module includes the following APIs of interest:

- A C++ version of the SAX2 parser⁶
- A C++ implementation of DOM, *The Document Object Model*⁷

SAX, which stands for Simple API for XML, is a low-level event-driven way of parsing XML. SAX can run on an XML file of any size.

DOM is a higher-level interface for dealing with XML elements as objects in a navigable tree structure. DOM loads the XML file into memory, so the maximum file size your application can handle is limited by the amount of available RAM.

14.2 Event-Driven Parsing

Working with SAX-style XML parsers means doing event-driven programming. The flow of execution depends entirely on the data that is being read from a file. This **inversion of control** means that the thread of execution will be more difficult to trace. Our code will be called by code inside the Qt library.

Invoking the parser involves creating a `reader` and a `handler`, hooking them up, and calling `parse()`, as shown in Example 14.4.

EXAMPLE 14.4 `src/xml/sax1/tagreader.cpp`

```
#include "structureparser.h"
#include <QFile>
#include <QXmlInputSource>
#include <QXmlSimpleReader>
#include <QDebug>
```

continued

⁶ <http://www.saxproject.org>

⁷ <http://www.w3c.org/DOM/>

```

int main( int argc, char **argv ) {
    if ( argc < 2 ) {
        qDebug() << QString("Usage: %1 <xmlfile>").arg(argv[0]);
        return 1;
    }
    for ( int i=1; i < argc; ++i ) {
        QFile xmlFile( argv[i] );
        QXmlInputSource source( &xmlFile );
        StructureParser handler;
        QXmlSimpleReader reader;
        reader.setContentHandler( &handler );
        reader.parse( source );
    }
    return 0;
}

```

- ❶ a custom derived instance of QXmlContentHandler
- ❷ the generic parser
- ❸ Hook up the objects together.
- ❹ Start parsing.

The interface for parsing XML is described in the abstract base class `QXmlContentHandler`. We call this a **passive interface** because these methods get called, just not from our code. `QXmlSimpleReader` is provided, which reads an XML file and generates parse events, calling methods on a content handler in response to them. Figure 14.1 shows the main classes involved.

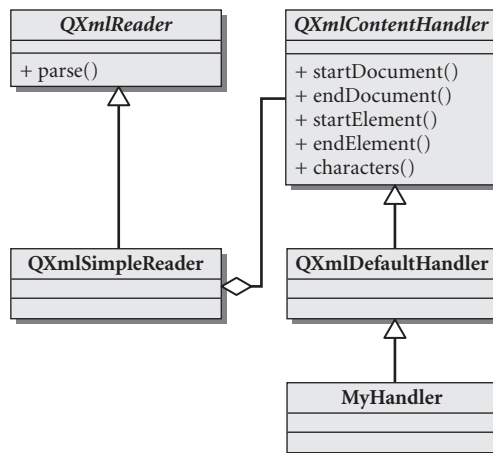


FIGURE 14.1 Abstract and concrete SAX classes

For the reader to provide any useful information, it needs an object to receive parse events. This object, a **parse event handler**, must implement a published interface, so it can “plug” into the parser, as shown in Figure 14.2.

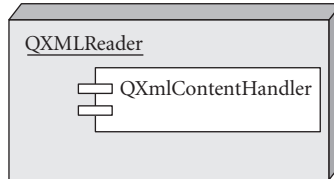


FIGURE 14.2 Plug-in component architecture

The handler derives (directly or indirectly) from `QXmlContentHandler`. The virtual methods get called by the parser when it encounters various elements of the XML file during parsing. This is *event-driven* programming: *You do not call these functions directly.*

EXAMPLE 14.5 src/xml/sax1/structureparser.h

```
#include <QXmlDefaultHandler>

class QString;

class MyHandler : public QXmlDefaultHandler {
public:
    bool startDocument();
    bool startElement( const QString & namespaceURI,
                      const QString & localName,
                      const QString & qName,
                      const QXmlAttributes & atts);
    bool characters(const QString& text);
    bool endElement( const QString & namespaceURI,
                    const QString & localName,
                    const QString & qName );

private:
    QString indent;
};
#endif
```

These passively called functions are often referred to as **callbacks**. They respond to events generated by the parser. The client code of `MyHandler` is the `QXmlSimpleReader` class, inside the Qt XML Module.



CONTENTHANDLER OR DEFAULTHANDLER?

`QXmlContentHandler` is an abstract class with many pure `virtual` methods, all of which must be overridden by any concrete derived class. Qt has provided a concrete class named `QXmlDefaultHandler` that implements the base class pure `virtual` methods as empty do-nothing bodies. You can think of this class as a concrete base class. Handlers derived from this class are not required to override all of the methods but must override some in order to accomplish anything.

If we do not properly override each handler method that will be used by our app, the corresponding `QXmlDefaultHandler` method, which does nothing, will be called instead. In the body of a handler function, you can

- Store the parse results in a data structure
- Create objects according to certain rules
- Print or transform the data in a different format
- Do other useful things

See Example 14.6.

EXAMPLE 14.6 `src/xml/sax1/myhandler.cpp`

```
[ . . . . ]
QTextStream cout(stdout, QIODevice::WriteOnly);

bool MyHandler::startDocument() {
    indent = "";
    return TRUE;
}

bool MyHandler::characters(const QString& text) {
    QString t = text;
    cout << t.remove('\n');
    return TRUE;
}

bool MyHandler::startElement( const QString&,
                              const QString&,
                              const QString& qName,
                              const QXmlAttributes& atts) {
    QString str = QString("\n%1\\%2").arg(indent).arg(qName);
    cout << str;
    if (atts.length()>0) {
        QString fieldName = atts.qName(0);
        QString fieldValue = atts.value(0);
        cout << QString("(%2=%3)").arg(fieldName).arg(fieldValue);
```

```

    }
    cout << "{";
    indent += "    ";
    return TRUE;
}

bool MyHandler::endElement( const QString&,
                           const QString& ,
                           const QString& ) {
    indent.remove( 0, 4 );
    cout << "}";
    return TRUE;
}
[ . . . . ]

```

The `QXmlAttributes` object passed into the `startElement()` function is a map, used to hold the *name = value* attribute pairs that were contained in the XML elements.

As it processes the file, the `parse()` function calls `characters()`, `startElement()`, and `endElement()` as these “events” are encountered in the file. In particular, each time a string of ordinary characters (between the beginning and end of a tag) is encountered, it’s passed as an array of bytes to the `characters()` function.

We ran the previous program on Example 14.3 and it transformed that document into Example 14.7, something that looks a little like LaTeX, another document format.

EXAMPLE 14.7 `src/xml/sax1/tagreader-output.txt`

```

\section(id=xmlintro){
  \title{ Intro to XML }
  \para{ This is a paragraph }
  \ul{
    \li{ This is an unordered list item. }
    \li(c=textbook){ This only shows up in the textbook } }
  \p{ Look at this example code below: }
  \include(src=xmlesamplecode.cpp){}

```

14.3 XML, Tree Structures, and DOM

The Document Object Model (DOM) is a higher-level interface for operating on XML documents. Working with and navigating through DOM documents is very straightforward (especially if you are familiar with `QObject` children).



SAX OR DOM? DOM requires the entire document to be loaded into memory, so it should not be used on large files. SAX, in contrast, can write or dispose of processed data while parsing and processing more data in a very large document.

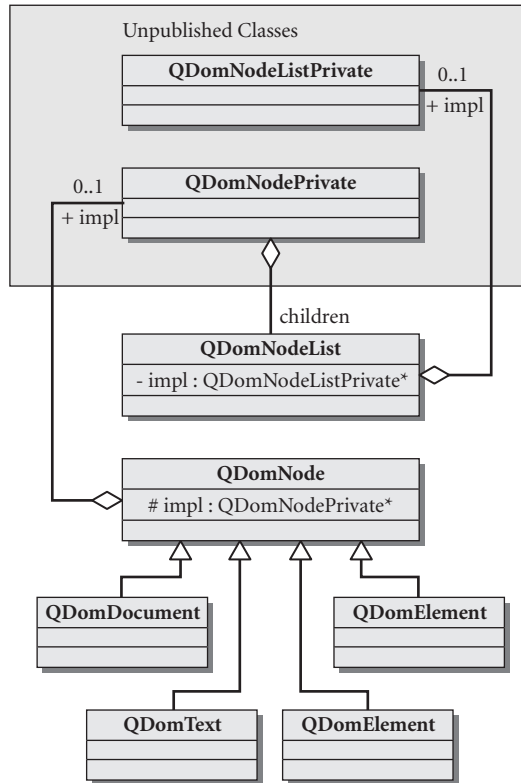


FIGURE 14.3 QDom UML model

The main classes in DOM are shown in Figure 14.3. `setContent()` parses the file, after which the `QDomDocument` contains a structured tree comprised of `QDomNode` objects. Each `QDomNode` could be an instance of `QDomElement`, `QDomAttr`, or `QDomText`. Every `QDomNode` has a parent except the root, which is a `QDomDocument`. Every node is reachable from the parent. DOM is another application of the Composite pattern.



The QDom classes are wrappers around private implementation classes. They contain no data except a pointer. This makes it possible to pass around QDomNode by value to other functions that can change the addressed objects (by adding attributes or children or by changing attributes). This gives QDom a more Java-like interface.

14.3.1 Visitor Pattern: DOM Tree Walking

Qt 4 provides full read/write access to trees of XML data. Nodes can be navigated through an interface which is similar to *but slightly different from* the QObject interface. Under the surface, SAX performs the parsing, and DOM defines a ContentHandler that creates the tree of objects in memory. All client code needs to do is call setContent(), and this causes the input to be parsed and the tree to be generated.

Example 14.8 transforms an XML document in place. After the tree is manipulated, it is serialized to a QTextStream where it will become savable/parsable again.

EXAMPLE 14.8 src/xml/domwalker/main.cpp

```
[ . . . . ]
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QString filename;
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " filename.xml" << endl;
        filename = "samplefile.xml";
    }
    else {
        filename = argv[1];
    }
    QFile f(filename);
    QString errorMsg;
    int errorLine, errorColumn;
    QDomDocument doc("SlackerDoc");
    bool result = doc.setContent(&f, &errorMsg,
        &errorLine, &errorColumn); 1
    QDomNode before = doc.cloneNode(true); 2
    Slacker slack(doc); 3
    QDomNode after = slack.transform(); 4
    cout << QString("Before: ") << before << endl;
    cout << QString("After: ") << after << endl;
}
```

continued

```

    QWidget * view = twinview(before, after); ❸
    view->show();
    app.exec();
    delete view;
}
[ . . . . ]

```

- ❶ Parse the file into a DOM tree, and store tree in empty doc.
- ❷ deep copy
- ❸ Send the tree to Slacker.
- ❹ Start the visitation.
- ❺ a pair of QTreeView objects separated by slider, using the QDomDocuments as models

The `Slacker` class is derived from `DomWalker`, another application of the Visitor pattern, specialized for walking through DOM trees.

Figure 14.4 shows the main classes used in this example.

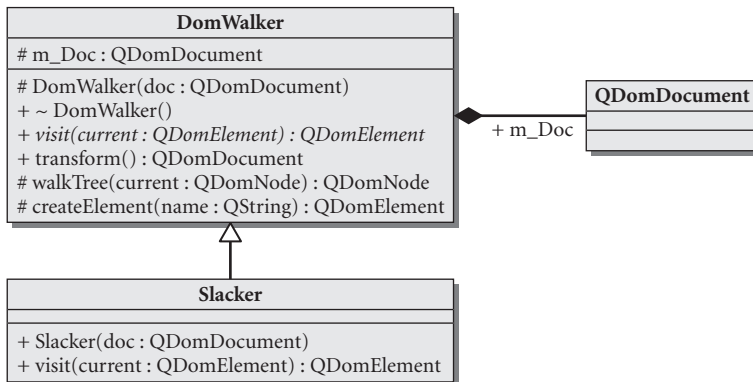


FIGURE 14.4 Domwalker and Slacker

EXAMPLE 14.9 `src/xml/domwalker/domwalker.cpp`

```

[ . . . . ]
QDomDocument DomWalker::transform() {
    walkTree(m_Doc);
    return m_Doc;
}

QDomNode DomWalker::walkTree(QDomNode current) {
    QDomNodeList dn1 = current.childNodes();
    for (int i=dn1.count()-1; i >=0; --i)
        walkTree(dn1.item(i));
}

```

❶


```

    if (current.nodeType() == QDomNode::ElementNode) {
        QDomElement ce = current.toElement();
        return visit(ce);
    }
    return current;
}
[ . . . . ]

```

- ❶ First process the children recursively.
- ❷ We only want to process elements, leaving all nodes unchanged.
- ❸ instead of a typecast

Notice that the `walkTree()` method, defined in Example 14.9, contains no pointers or typecasts. The `QDom(Node|Element|Document|Attribute)` types are smart-pointers. We “downcast” from `QDomNode` to `QDomElement`, or `QDomXXX`, using `QDomNode::toElement()` or `toXXX()` conversion functions.



When traversing a tree, it is possible to use only the `QDomNode` interface, but when operating on an actual XML element, “casting” down to `QDomElement` adds some convenient functions for dealing with the Element as a whole, as well as its attributes (which are themselves `QDomNode` child objects).

Even though in this example `QDomNode/QDomElement` objects are being passed and returned by value, it is still possible to change the underlying DOM objects through the temporary copies. Through interface trickery, `QDom` objects look and feel like Java-style references, and hold pointers inside, rather than actual data.

`Slacker` defines how to transform documents from one XML format to another. It is an extension of `DomWalker` and overrides just one method, `visit()`. Defined in Example 14.10, this method has a special rule for each kind of element.

EXAMPLE 14.10 `src/xml/domwalker/slacker.cpp`

```

[ . . . . ]
QDomElement Slacker::visit(QDomElement element) {
    QString name = element.tagName();
    [ . . . . ]
    /* Mapping elements: */
    if (name == "p") {
        element.setTagName("para");
        return element;
    }
}

```

continued

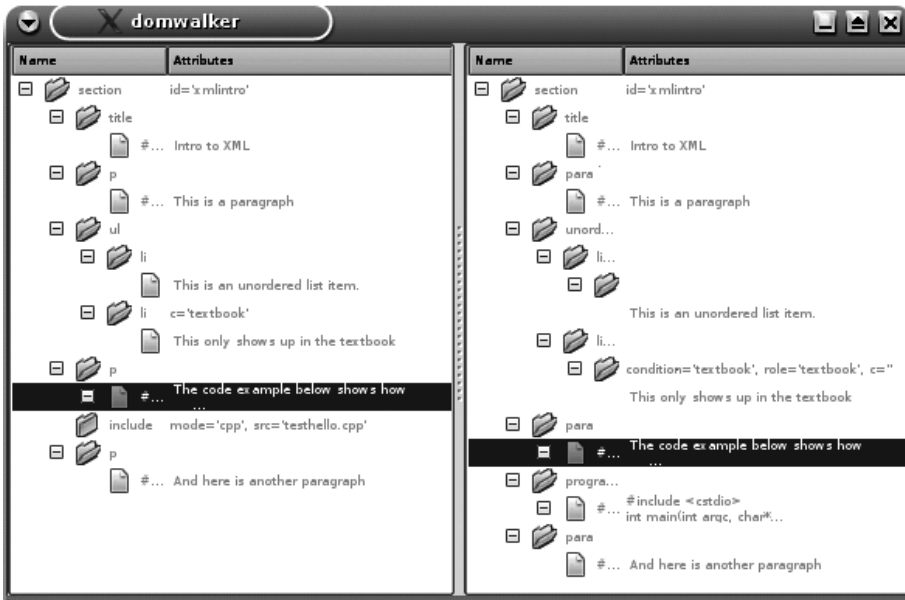
```

if (name == "ul") {
    element.setTagName("unorderedlist");
    return element;
}
if (name == "b") {
    element.setAttribute("role", "strong");
    element.setTagName("emphasis");
    return element;
}
/* This transformation is more interesting because we
   replace <li> text </li> with
   <listitem><para> text </para></listitem>
*/
if (name == "li") {
    QDomNode parent = element.parentNode();
    QDomElement listitem = createElement("listitem");
    parent.replaceChild(listitem, element); /*
        remove the li tag, but put a listitem in its place */
    element.setTagName("para"); ❶
    listitem.appendChild(element);
    return listitem;
}[ . . . ]

```

❶ Recall element is the original text node.

When we run this example, it pops up a tree view of the before/after XML documents side by side, so we can inspect them, as the next screenshot shows.



14.3.2 Generation of XML with DOM

DOM documents are normally created by parsers to represent XML from an input stream, but DOM can also be used to generate XML structures as output. It is preferable to generate XML through an API rather than by printing formatted strings because DOM generation guarantees that the resulting document will be parsable again.

In Figure 14.5, `DocbookDoc` is a factory for `QDomElements`. It is derived from `QDomDocument`, and specialized for creating Docbook/XML documents.

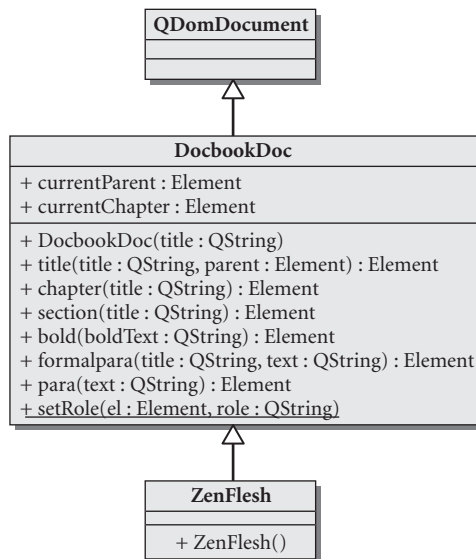


FIGURE 14.5 `DocbookDoc`

In the header file for this class, excerpted in Example 14.11, we added a `typedef` to improve readability. In the DOM standard, all DOM classes are simply called `Node`, `Element`, `Document`, and `Attribute`.

EXAMPLE 14.11 `src/libs/docbook/docbookdoc.h`

```
typedef QDomElement Element; ❶
```

❶ Saves on typing and is consistent with Java DOM.

We can build documents by creating chapters, sections, and paragraphs, as we see in Example 14.12.

EXAMPLE 14.12 `src/xml/xmlbuilder/zenflesh.cpp`

```

#include <qstd.h>
using namespace qstd;
#include "../docbook.h"

class ZenFlesh : public DocbookDoc {
public: ZenFlesh();
};

ZenFlesh::ZenFlesh() :
    DocbookDoc("Zen Flesh, Zen Bones") {

    chapter("101 Zen Stories");
    section("A cup of tea");
    para("Nan-in served a cup of tea.");
    section("Great Waves");
    QDomElement p = para("o-nami the wrestler sat in meditation"
        "and tried to imagine himself as a bunch of great waves.");
    setRole(p, "remark");
    chapter("The Gateless Gate");
    formalpara("The Gateless Gate",
        "In order to enter the gateless gate, you must have a ");
    bold(" mindless mind.");

    section("Joshu's dog");
    para("Has a dog buddha nature or not?");

    section("Haykujo's Fox");
    QDomElement fp = formalpara("This is a special topic",
        "Which should have a role= remark attribute");
    setRole(fp, "remark");
}

int main() {
    ZenFlesh book;
    cout << book << endl;
}

```

The constructor generates a little book in XML, which after pretty-printing, could look like Example 14.13.

EXAMPLE 14.13 `src/xml/zen.xml`

```

<book>
  <title>Zen Flesh, Zen Bones</title>
  <chapter>
    <title>101 Zen Stories</title>
    <section>
      <title>A cup of tea</title>

```

```

        <para>Nan-in served a cup of tea.</para>
    </section>
    <section>
        <title>Great Waves</title>
        <para>
            o-nami the wrestler sat in meditation and tried
            to imagine himself as a bunch of great waves.
        </para>
    </section>
</chapter>
<chapter>
    <title>The Gateless Gate</title>
    <formalpara>
        <title>The Gateless Gate</title>
        In order to enter the gateless gate,
        you must have a <emphasis role="strong">
            mindless mind</emphasis>
    </formalpara>
    <section>
        <title>Joshu's dog</title>
        <para>Has a dog buddha nature or not?</para>
    </section>
    <section>
        <title>Haykujo's Fox</title>
        <formalpara role="remark">
            <title>This is a special topic</title>
            Which should have a role="remark" attribute
        </formalpara>
    </section>
</chapter>
</book>

```

The advantage of this format is that it can be easily converted into HTML (or PDF, or LaTeX) using `xsltproc` and the Docbook/XSL stylesheets [docbookxsl]. Example 14.14 shows the invocation for generating an HTML version.

EXAMPLE 14.14 `src/xml/zen2html`

```

#!/bin/sh
# translates zen.xml into index.html
# requires gnu xsltproc and docbook-xsl
xsltproc ../../../../docbook-xsl/html/onechunk.xsl zen.xml

```

Now let's look at Example 14.15, where the elements are created. Each major Docbook language element has a corresponding factory method in `DocbookDoc`.

EXAMPLE 14.15 `src/libs/docbook/docbookdoc.cpp`

```
[ . . . . ]

DocbookDoc::DocbookDoc(QString titleString) {
    Element root = createElement("book");
    appendChild(root);
    title(titleString, root);
    currentParent = root;
}

Element DocbookDoc::bridgehead(QString titleStr) {
    Element retval = createElement("bridgehead");
    Element titleEl = title(titleStr);
    currentParent.appendChild(retval);
    return retval;
}

Element DocbookDoc::title(QString name, Element parent) {
    Element retval = createElement("title");
    QDomText tn = createTextNode(name);
    retval.appendChild(tn);
    if (parent != Element())
        parent.appendChild(retval);
    return retval;
}

Element DocbookDoc::chapter(QString titleString) {
    Element chapter = createElement("chapter");
    title(titleString, chapter);
    documentElement().appendChild(chapter);
    currentParent = chapter;
    currentChapter = chapter;
    return chapter;
}

Element DocbookDoc::para(QString textstr) {
    QDomText tn = createTextNode(textstr);
    Element para = createElement("para");
    para.appendChild(tn);
    currentParent.appendChild(para);
    currentPara = para;
    return para;
}
```

In addition, we have some character-level elements that only modify text, shown in Example 14.16.

EXAMPLE 14.16 `src/libs/docbook/docbookdoc.cpp`

```
[ . . . . ]

Element DocbookDoc::bold(QString text) {
    QDomText tn = createTextNode(text);
    Element emphasis = createElement("emphasis");
    setRole(emphasis, "strong");
    emphasis.appendChild(tn);
    currentPara.appendChild(emphasis);
    return emphasis;
}

void DocbookDoc::setRole(Element el, QString role) {
    el.setAttribute("role", role);
}
}
```

Because each `QDomNode` must be created by `QDomDocument`, it makes sense to extend `QDomDocument` to write our own DOM factory.

`DocbookDoc` adds its newly created Elements as children to previously created Elements, depending on what kind of Element is being created.

**EXERCISE: XML, TREE STRUCTURES,
AND DOM**

Rewrite the Slacker transformer tree-walker so that instead of modifying a DOM tree in place, it creates a new DOM tree using an Element factory derived from a `QDomDocument`. You can use the `DocbookDoc` class as a starting point.

REVIEW QUESTIONS

1. If there is a syntax error in your XML file, how do you determine the cause and location?
2. SAX is an event-driven parser. What kinds of events does it respond to?
3. Qt (as well as other language frameworks) offers two XML parser APIs, one is SAX and the other is DOM. Compare and contrast them. Why would you use one rather than the other?
4. If you have a `QDomNode` and it is actually “pointing” to a `QDomElement`, how do you get a reference to the `QDomElement`?
5. Explain how `DomWalker` is an application of the Visitor pattern.

15

CHAPTER 15

Meta Objects, Properties, and Reflective Programming

In this chapter we introduce the idea of reflection, the self-examination of an object's members. Using reflective programming, it becomes possible to write general-purpose operations that work on classes of varied structures. Using `QVariant`, a generic value-holder, we can operate on built-in types as well as other common types in a uniform way.

15.1 Anti-patterns	342
15.2 QMetaObject: The MetaObject Pattern	344
15.3 Type Identification and qobject_cast	345
15.4 Q_PROPERTY Macro: Describing QObject Properties	347
15.5 QVariant Class: Accessing Properties	350
15.6 DataObject: An Extension of QObject	353
15.7 Property Containers: PropsMap	355



15.1 Anti-patterns

Anti-pattern¹ is a term first coined by [Gamma95] to mean a common design pitfall. An anti-pattern is called this because many design patterns are designed to avoid such pitfalls.

In other words, design patterns are commonly used solutions to anti-patterns, which are commonly faced problems. Some examples of anti-patterns are:

- Copy and paste programming: Copying and modifying existing code without creating more generic solutions.
- Hard coding: Embedding assumptions about the environment (such as constant numbers) in multiple parts of the software
- Interface bloat: Having too many methods, or too many arguments in functions; in general, refers to a complicated interface that is hard to reuse or implement
- Reinventing the (square) wheel: Implementing some code when something (better) already exists in the available APIs
- God Object: An object that has too much information or too much responsibility. This can be the result of having too many functions in a single class. It can arise from many situations, but often happens when code for a model and view are combined in the same class.

In Figure 15.1, `customer` includes member functions for importing and exporting its individual data members in XML format. `getWidget()` provides a special GUI widget that the user can use to enter data from a graphical application. In addition, there are custom methods for input/output via `iostream`.

This class is a model, because it holds onto data and represents some abstract entity. However, this class also contains view code, because of the `createWidget()` method. In addition, it contains serialization code specific to the data type. That is

¹ <http://en.wikipedia.org/wiki/Anti-pattern>

Customer
- name : QString - address : QString - city : QString - birthdate : QDate
+ setName(newName : QString) + setAddress(newAddress : QString) + setCity(newCity : QString) + setBirthdate(newDate : const QDate&) + getName() : QString + getAddress() : QString + getCity() : QString + getBirthdate() : QDate + exportXML(os : ostream&) + importXML(is : istream&) + createWidget() : QWidget* + getWidget() : QWidget*

```
operator<<(ostream& out, const Customer& cust) : ostream&
operator>>(istream& in, Customer& cust): istream&
```

FIGURE 15.1 Anti-pattern

too much responsibility for a data model. It is quickly becoming an example of the God Object anti-pattern.

As we add other data model classes (Address, ShoppingCart, Catalog, CatalogItem, etc.), each of them also would need these methods:

- createWidget ()
- importXML ()
- exportXML ()
- operator<< ()
- operator>> ()

This could lead to the use of copy-and-paste programming, another anti-pattern.

If we ever change the data structure, corresponding changes would need to be made to all presentation and I/O methods. Bugs introduced when maintaining this code are very likely.

If `Customer` were **reflective**, meaning that it had the ability to determine useful things about its own members (e.g.: How many properties? What are their names? What are their types? How do I load/store them? What are the child objects?), then we could define a generic way to read and write objects that would work for `Customer` and any other similarly reflective class.

15.2 QMetaObject: The MetaObject Pattern

By abstracting the abstract data type itself, we achieve what is called a **MetaObject**. A `MetaObject` is an object that describes the structure of another object.²

The MetaObject Pattern

The `QMetaObject` is Qt's implementation of the `MetaObject` pattern. It provides information about properties and methods of a `QObject`. The `MetaObject` pattern is sometimes known as the `Reflection` pattern.

A class that has a `MetaObject` supports **reflection**. This is a feature found in many object-oriented languages. It does not exist in C++, but Qt's `MetaObject` compiler (`moc`) generates the code to support this for desired classes.

As long as certain conditions apply,³ each class derived from `QObject` will have a `QMetaObject` generated for it by `moc`, as shown in Figure 15.2. `QObject` has a member function that returns a pointer to the object's `QMetaObject`.

```
QMetaObject* QObject::metaObject () const [virtual]
```

A `QMetaObject` can be used to invoke functions such as:

- `className()`, which returns the class name as a `const char*`
- `superClass()`, which returns a pointer to the `QMetaObject` of the base class if there is one (or 0 if there is not)
- `methodCount()`, which returns the number of member functions of the class
- Several other useful functions that we will discuss in this chapter

The signal and slot mechanism also relies on the `QMetaObject`.

By using the `QMetaObject` and `QMetaProperty`, it is possible to write code that is generic enough to handle all self-describing classes.

² *Meta*, the latin root meaning *about*, is used for its literal definition here.

³ Each class must be defined in a header file, listed in the project file's `HEADERS`, and must include the `Q_OBJECT` macro in its class definition.

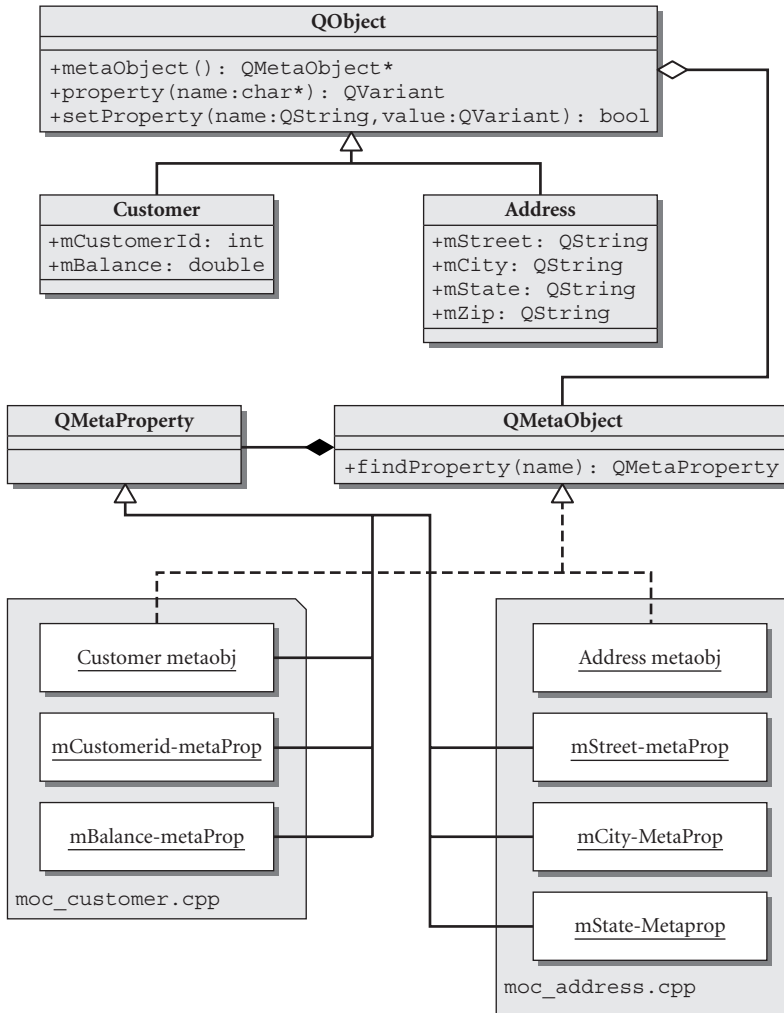


FIGURE 15.2 MetaObjects

15.3 Type Identification and `qobject_cast`

RTTI, or **Run-Time Type Identification**, as its name suggests, is a system for determining at runtime the actual type of an object, to which we may only have a base class pointer.

In addition to C++'s RTTI operators, `dynamic_cast` and `typeid` (Section 19.8), Qt provides two mechanisms for run-time type identification.

1. `qobject_cast`
2. `QObject::inherits()`

`qobject_cast` is an ANSI-style typecast operator (Section 19.7). ANSI typecasts look a lot like template functions:

```
DestType* qobject_cast<DestType*> ( QObject * qoptr )
```

A **typecast** operator converts an expression from one type to another, following certain rules and restrictions imposed by the types and the language. Like other cast operators, `qobject_cast` takes the destination type as a template parameter. It returns a different-typed pointer to the same object. If at runtime the actual pointer type cannot be converted to `DestType*`, then the conversion fails and the value returned is `NULL`.

As the signature suggests, `qobject_cast` is type-restricted to arguments of type `DestType*`, where `DestType` is derived from `QObject` and the class was fully processed by `moc`. Therefore, `qobject_cast` is actually a **downcast** operator, similar to `dynamic_cast`.

In situations where you have base class pointers to derived class objects, down-casting makes it possible to call derived class methods that do not exist in the base class interface. In Example 15.1, we take advantage of the fact that `QApplication`, and `MyApplication`, both derive from `QObject`.

EXAMPLE 15.1 `src/qtrtti/myapp-classdef.cpp`

```
class MyApplication : public QApplication {
    Q_OBJECT /* Required for Qt RTTI */
public:
    static MyApplication* instance();
    QString imageURL() const;
    [... other members ...]
};

MyApplication* MyApplication::instance() {
    static MyApplication* inst = 0;
    if (inst == 0) {
        inst = qobject_cast<MyApplication*>(qApp);
    }
    return inst;
}
```

Because `qApp` always points to the currently running `QApplication`, this function will return non-zero only if a `MyApplication` is the current running application. The downcast operation, which some say is expensive, happens *only once*, to ensure a properly typed `MyApplication` pointer. Future calls to `instance()` will return the previously cast pointer, avoiding repeated calls to expensive runtime checking operations. Now it becomes possible to obtain the properly typed `MyApplication` instance from other locations in the code:

```

QString imagePath(QString filename) {
    MyApplication* app = MyApplication::instance();
    QString path = app->imagesURL() + "/" + filename; ❶
    return path;
}

```

❶ The same file separator char works on *all* operating systems that support Qt!



The implementation of `qobject_cast` makes no use of C++ RTTI. The code for this operator is generated by the MetaObject Compiler (`moc`).

`QObject` also offers a Java-style typechecking function, `inherits()`. Unlike `qobject_cast`, `inherits()` accepts a `char*` type name instead of a type expression. This operation is slower than `qobject_cast` because it requires an extra hashtable lookup, but it can be useful if you need input-driven type checking.

Example 15.2 shows some client code that uses `inherits()`.

EXAMPLE 15.2 `src/qtrtti/qtrtti.cpp`

```

[ . . . . ]

// QWidget* w = &s; ❶

if (w->inherits("QAbstractSlider")) cout << "Yes, it is ";
else cout << "No, it is not";
cout << "a QAbstractSlider" << endl;

if (w->inherits("QListView")) cout << "Yes, it is ";
else cout << "No, it is not ";
cout << "a QListView" << endl;

return 0;
}

```

❶ pointer to some widget

15.4 Q_PROPERTY Macro: Describing QObject Properties

The property facility gives us a choice of ways to access data members:

- Directly, through the classic getters/setters (faster, more efficient)
- Indirectly, through the `QObject/QMetaObject` interface (more generic)

EXAMPLE 15.3 src/properties/customer-props.h

```

[ . . . . ]
class CustProps : public QObject {
    Q_OBJECT ❶

    /* Each property declaration has the following syntax:

    Q_PROPERTY( type name READ getFunction [WRITE setFunction]
    [RESET resetFunction] [DESIGNABLE bool]
    [SCRIPTABLE bool] [STORED bool] )
    */

    Q_PROPERTY( QString Id READ getId WRITE setId );
    Q_PROPERTY( QString Name READ getName WRITE setName );
    Q_PROPERTY( QString Address READ getAddress WRITE setAddress );
    Q_PROPERTY( QString Phone READ getPhone WRITE setPhone);
    Q_PROPERTY( QDate DateEstablished
                READ getDateEstablished
                WRITE setDateEstablished );
    Q_PROPERTY( CustPropsType Type READ getType WRITE setType );
    Q_ENUMS( CustPropsType ) ; ❷

public:
    enum CustPropsType
    { Corporate, Individual, Educational, Government }; ❸

    CustProps(QObject *parent = 0, const QString name = QString());

    QString getId() const {
        return m_Id;
    }
[ . . . . ]
    CustPropsType getType() const {
        return m_Type;
    }

    QString getTypeString() const;
    void setId(const QString &newId);
[ . . . . ]
    // Overloaded, so we can set the type 2 different ways:
    void setType(CustPropsType newType);
    void setType(QString newType);
private:
    QString m_Id, m_Name, m_Address, m_Phone;
    QDate m_Date;
    CustPropsType m_Type;
};
[ . . . . ]

```

❶ macro required for moc to preprocess class

❷ special macro to generate string-to-enum conversion functions

❸ The enum type definition must be in the same class definition as the Q_ENUMS macro.

In Example 15.3, we have a customer class with a Qt property defined for each data member. The name of a property must not be the same as any data member name. We have adopted the common practice of giving each property that corresponds to a data member a name that is based on the corresponding data member name. If the data member is `m_DataItem`, the corresponding property is named `DataItem`. We discuss an example of a class that has properties that do not correspond to data members in Section 16.3.3.

Notice the `enum CustPropsType` that is defined in the public section of the class `CustProps`. Just above that definition, the `Q_ENUMS` macro tells `moc` to generate some functions for this property in the `QMetaProperty` to aid in string conversions for enum values.

The setters and getters are defined in Example 15.4. They are implemented in the usual way.

EXAMPLE 15.4 `src/properties/customer-props.cpp`

```
[ . . . . ]
CustProps::CustProps(QObject *parent, const QString name)
    :QObject(parent) {
    setObjectName(name);
}

void CustProps::setId(const QString &newId) {
    m_Id=newId;
}
[ . . . . ]
void CustProps::setType(CustPropsType theType) {
    m_Type=theType;
}

/* Method for setting enum values from Strings. */
void CustProps::setType(QString newType) {
    static const QMetaObject* meta = metaObject();
    static int propindex = meta->indexOfProperty("Type");
    static const QMetaProperty mp = meta->property(propindex);

    QMetaEnum memum = mp.enumerator();
    const char* ntyp = newType.toAscii().data();
    m_Type = static_cast<CustPropsType>(memum.keyToValue(ntyp));
}

QString CustProps::getTypeString() const {
    return property("Type").toString();
}
[ . . . . ]
```

- ❶ Overloaded version that accepts a string as an argument. Sets value to `-1` if unknown.
 - ❷ Because they are static locals, the initializations happen only once.
 - ❸ This code gets executed each time.
-

The implementation of the overloaded function `setType(QString)` takes advantage of `QMetaProperty`'s `Q_ENUM` macro to convert the `QString` to the proper enumerated value. To obtain the correct `QMetaProperty` object for an enum, we first get the `QMetaObject` and call functions `indexOfProperty()` and `property()` to find it. `QMetaProperty` has a function called `enumerator()` that you can use to convert strings to enums. If the given `QString` argument does not match one of the enumerators, the `keyToValue()` function will return the value `-1`.

Static Local Variables

Observe that we have declared the three local (block scope) variables, `meta`, `propindex`, and `mp`, to be `static`.

Each call to this function will require the same `QMetaProperty` object, so there is no need to have repeated calls to the `QMetaObject` functions require the iteration each time. `static` local variables are initialized only once, which is our intention—repeated calls to this function will use the same `QMetaProperty` object to do the conversion. Using `static` local variables this way in a function can greatly improve the run-time performance for that function.⁴

15.5 QVariant Class: Accessing Properties

One frequently encountered problem C++ developers have as they try to make things more object oriented, arises from the fact that primitive types of C++ (e.g., `int`, `float`, `char*`, etc.) do not derive from a common base class. They're called **primitive** because they're smaller, simpler, and used to compose more complex things.

We would like to be able to retrieve the value of any property through the following function:

```
[returntype] QObject::property(QString propertyName);
```

Templates are one way to address this issue—template functions cause code to be generated for each used type. In Qt, we have another object-oriented way to solve the problem: through the use of `QVariant`.

⁴ This depends on how expensive creating the objects are and how often the function is called.

`QVariant` is a union wrapper⁵ for all the basic types, as well as all permitted `Q_PROPERTY` types. You can create a `QVariant` as a wrapper around another typed value. It remembers its type, and has member functions for getting and setting its value.

`QVariant` has a rich interface for data conversion and validity checking. In particular, there is a `toString()` function that returns a `QString` representation for its different types. This class greatly simplifies the property interface.

Example 15.5 shows some client code for the `CustProps` class defined in Example 15.3.

EXAMPLE 15.5 `src/properties/testcustomerprops.cpp`

```
[ . . . . ]

int main() {
    CustProps cust;
    cust.setName("Falafal Pita");
    cust.setAddress("41 Temple Street; Boston, MA; 02114");
    cust.setPhone("617-555-1212");
    cust.setType("Government");
    ASSERT_EQUALS(cust.getType(), CustProps::Government);
    QString originalid = "834";
    cust.setId(originalid);
    QVariant v = cust.property("Id");
    QString str = v.toString();
    ASSERT_EQUALS(originalid, str);
    QDate date(2003, 7, 15);
    cust.setProperty("DateEstablished", QVariant(date));
    QDate anotherDate = cust.getDateEstablished();
    ASSERT_EQUALS(date, anotherDate);
    cust.setId(QString("anotherId"));
    qDebug() << objToString(&cust);
    cust.setType(CustProps::Educational);
    qDebug() << " Educational=" << cust.getType();
    cust.setType("BogusType");
    qDebug() << " Bogus=" << cust.getType();
    return 0;
}
```

continued

⁵ A **union** is a struct that declares several data members that are all allocated at the same address. This means that the union will occupy only enough memory to accommodate the largest of the declared data members. When instantiated, a union can only store a value for one of the declared members.

- ❶ setting some simple properties
- ❷ setting enum property as a string
- ❸ comparing to enum value
- ❹ setting a string property
- ❺ getting it back as a QVariant through the QObject base class method
- ❻ setting date properties, wrapped in QVariants
- ❼ The date comes back through the type-specific getter.

In Example 15.6, we show a reflective `objToString()` method that works on any class with Qt properties defined. It works by iterating through each `property()` value, in a way that is comparable to the `java.lang.reflect` interface.

EXAMPLE 15.6 `src/properties/testcustomerprops.cpp`

```
[ . . . . ]

QString objToString(const QObject* obj) {
    QStringList result;
    const QMetaObject *meta = obj->metaObject(); ❶
    result += QString("class %1 : public %2 {")
        .arg(meta->className())
        .arg(meta->superClass()->className());
    for (int i=0; i < meta->propertyCount(); ++i) {
        const QMetaProperty qmp = meta->property(i);
        result += QString("  %1 %2 = %3;")
            .arg(qmp.type())
            .arg(qmp.name())
            .arg(obj->property(qmp.name()).toString());
    }
    result += "};";
    return result.join("\n");
}
```

- ❶ We introspect into the object via the `QMetaObject`.

The program outputs an object's state in a C++-style format:

```
src/properties> ./properties
class CustProps : public QObject {
    10 objectName = ;
    10 Id = anotherId;
    10 Name = Falafal Pita;
    10 Address = 41 Temple Street; Boston, MA; 02114;
    10 Phone = 617-555-1212;
    14 DateEstablished = 2003-07-15;
    2 Type = 3;
};
Educational= 2
Bogus= -1
```

15.6 DataObject: An Extension of QObject

We have developed an extension to `QObject` named `DataObject` that we can use as a base class for other data types that require any of the following features:

- A virtual interface for obtaining properties, `MetaProperties`, and `metaClass` information
- Convenience functions for copying and comparing property values of `DataObjects`
- A `toString()` function that returns a presentation of all the properties of the object in XML format

This improved interface makes `DataObject` a Façade for `QObject`.

`DataObject`, shown in Figure 15.3, is used in several forthcoming examples to demonstrate various design patterns.

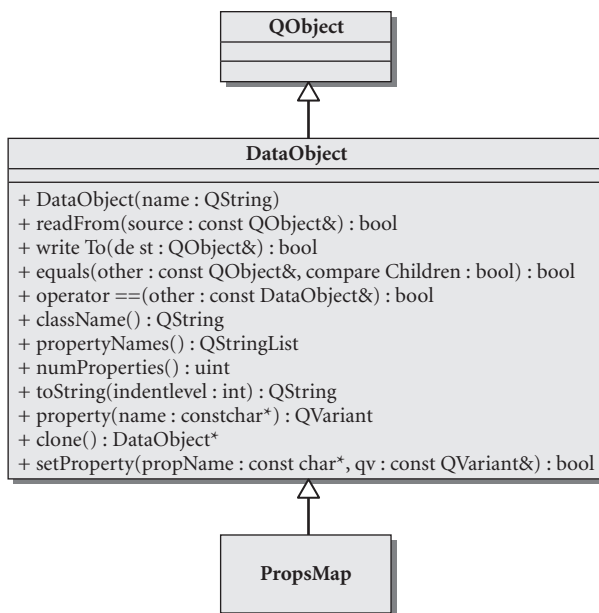


FIGURE 15.3 `DataObject`

`DataObject` is Qt-property aware and takes advantage of this interface for reading and writing properties of arbitrary `QObject`s, as shown in Example 15.7.

EXAMPLE 15.7 `src/libs/dataobjects/dataobject.cpp`

```
[ . . . . ]

bool DataObject::readFrom(const QObject& source) {
    bool retval = true;
    const QMetaObject* meta = source.metaObject();
    int count = meta->propertyCount();
    for (int i=0; I<count; ++i) {
        QMetaProperty metap = meta->property(i);
        const char* pname = metap.name();
        if (metap.isWritable()) {
            retval = setProperty(pname, source.property(pname))
                && retval;
        }
    }
    return retval;
}

[ . . . . ]

bool DataObject::writeTo(QObject& dest) const {
    bool result = true;
    foreach (QString propname, propertyNames()) {
        if (metaProperty(propname).isWritable()) {
            QVariant val = property(propname);
            result = dest.setProperty(propname.toAscii(), val)
                && result;
        }
    }
    return result;
}

[ . . . . ]
```

**EXERCISES: META OBJECTS, PROPERTIES,
AND REFLECTIVE PROGRAMMING**

1. Many people have several pets, each of which require periodic maintenance (visits to the veterinarian, immunizations, vitamins, grooming, etc.).
 - Derive a class named `Pet` from `DataObject`. Supply this class with appropriate attributes that can uniquely describe the pet (type, breed, name, ID, birthday, etc.). For each attribute, you will need a getter and a setter so that you can set up a `Q_PROPERTY`.
 - Derive another class from `DataObject` that you can call `Maintenance` (if you can't think of a better name). This class should have attributes that uniquely describe a particular maintenance event (type of event, date, cost, etc.). `Maintenance` objects will be stored as children of `Pet` objects.
 - Derive a `PetList` class from `DataObject`. `Pet` objects will be children of `PetList`.

- Serialization is facilitated by the `DataObject::toString()` function. Find out how to use this for your `PetWriter` class and how to reverse the action in your `PetReader` class.
 - Design a nice GUI for these classes so that the user can enter, store, and display data.
2. Do the exercise in Section 25.1.

15.7 Property Containers: PropsMap

Some objects, such as `FileTagger` (see Section 16.3.3) are **heavyweight objects**, meaning that creating/copying them requires allocation of external resources (the `ID3_tag` object, specifically). If you only wish to capture the object's properties without allocating another external resource, this is a form of in-memory serialization.

`PropsMap` has its name because it uses a map as a container for name/property pairs, Perl-style.⁶ Example 15.8 defines the class, using a `QMap` to store the properties.

EXAMPLE 15.8 `src/libs/dataobjects/propsmap.h`

```
[ . . . . ]
class PropsMap : public DataObject {
    Q_OBJECT
public:
    PropsMap(QString classn=QString()): m_ClassName(classn) {}
    PropsMap(const QObject& other);
    virtual QString className() const;
    bool readFrom(const QObject& source);
    QVariant property(QString key) const {
        return m_ValueMap[key];
    }
    virtual QStringList propertyNames() const {
        return m_ValueMap.keys();
    }
public slots:
    virtual bool setProperty(const QString& key,
                            const QVariant& value) ;
    virtual bool setProperty(const QString& key,
                            const QString& value) ;
};
```

continued

⁶ In perl, each “object” is simply a hashtable of key/value pairs with methods on it. All an object's data members are stored in the hash table.

```
private:
    QMap<QString, QVariant> m_ValueMap; ❶
    QString m_ClassName;                ❷
};
[ . . . ]
```

❶ holder of properties

❷ the pseudo-class that the object belongs to

The implementation of these methods, shown in Example 15.9, is straightforward.

EXAMPLE 15.9 `src/libs/dataobjects/propsmap.cpp`

```
#include <qmetaobject.h>
#include <qvariant.h>
#include "propsmap.h"

PropsMap::PropsMap(const QObject& other) {
    readFrom(other);
}

bool PropsMap::readFrom(const QObject& source) {
    m_ClassName =source.metaObject()->className();
    return DataObject::readFrom(source);
}

QString PropsMap::className() const {
    if (m_ClassName != QString())
        return m_ClassName;
    else
        return "PropsMap";
}

bool PropsMap::setProperty(const QString& key,
                           const QString & value) {
    return setProperty(key, QVariant(value));
}

bool PropsMap::setProperty(const QString& key,
                           const QVariant& value) {
    m_ValueMap[key]=value;
    return true;
}
```

As you examine the code in `libdataobject`, you will find that `PropsMap` is used for the following purposes:

1. As a default type to return from `ObjectFactory` when nothing is known about the given classname
2. As a temporary container of property information from `FileTagger` objects

REVIEW QUESTIONS

1. What is an anti-pattern? Give two examples.
2. How do you determine the number of properties defined in a `QObject`?
3. How does the `QMetaObject` code for each of your `QObject`-derived classes get generated?
4. What is a downcast? In what situations do we use them?
5. What are the advantages of defining properties over regular getters and setters?
6. What does the `property()` function return? How do we obtain the actual stored value?

16

CHAPTER 16

More Design Patterns

In this chapter, we present design patterns from each of the three categories: creational, structural, and behavioral.

16.1 Creational Patterns	360
16.2 Serializer Pattern Revisited	373
16.3 The Façade Pattern	381



16.1 Creational Patterns

By using **creational patterns** to manage object creation, we gain flexibility that makes it possible to choose or change the kinds of objects created or used at runtime, and also to manage object deletion automatically. Especially in large software systems, managing the creation of objects is important for flexibility in program design, maintaining a separation between layers of code and ensuring that objects are properly deleted when they are no longer needed.

In C++, a **factory** is a program component, generally a class, that is responsible for creating objects. The idea of a factory is to separate object creation from object usage.

A factory class generally has a function that obtains dynamic memory for the new object and returns a base class pointer to that object. This approach allows new derived types to be introduced without necessitating changes to the code that uses the created object.

We will discuss several design patterns that use factories and show examples of these design patterns.

When the responsibility for heap object creation is delegated to a `virtual` function, we call this a **Factory method**.

By making the factory method pure `virtual` and writing concrete derived factory classes, this becomes an **abstract factory**.

The **Abstract Factory pattern** provides an interface for defining factories that share abstract features but differ in concrete details. Client code can instantiate a particular subfactory and then use the abstract interface to create objects.

By imposing **creation rules** that prevent direct instantiation of a class, we can force clients to use factory methods to create all instances. When we combine creation rules with a factory method that always returns the same singleton object, this is an example of the **Singleton pattern**.

The **Singleton pattern** restricts a class so that only one instance can be created. This can be accomplished by making its constructor `private` or `protected` and providing an `instance()` function that creates a new instance if one does not already exist, but returns a pointer or reference to that instance if it does.

Consider two ways of creating a `Customer`:

```
Customer* c1 = new Customer(name);
Customer* c2 = CustomerFactory::instance()->newCustomer(name)
```

In the first case, we are hard-coding the class name and calling a constructor directly. The object will be created in memory using the default heap storage. Hard-coded class names in client code can limit the reusability of the code.

In the second case, we create a `Customer` object indirectly using a factory method called `newCustomer()`. In fact, `newCustomer()` is being called on a singleton `CustomerFactory` instance. `CustomerFactory` is a singleton because only one instance is available to us, via a static factory method, `instance()`.

16.1.1 Abstract Factory

`AbstractFactory`, defined in Example 16.1, is a simple class.

EXAMPLE 16.1 `src/libs/dataobjects/abstractfactory.h`

```
[ . . . . ]
class AbstractFactory {
public:
    virtual DataObject* newObject (QString className) = 0;
    virtual ~AbstractFactory() {}
};
[ . . . . ]
```

The `newObject()` method is pure `virtual`, so it must be overridden in derived classes. Example 16.2 shows a concrete class derived from `AbstractFactory`.

EXAMPLE 16.2 `src/libs/dataobjects/objectfactory.h`

```
[ . . . . ]
class ObjectFactory : public QObject, public AbstractFactory {
    Q_OBJECT
public:
    static ObjectFactory* instance() ;
    virtual DataObject* newObject (QString className);
    Address* newAddress (Country::CountryType country =
        Country::Undefined);
    Address* newAddress (QString countryName = "USA");
protected:
    ObjectFactory() {}
};
[ . . . . ]
```

`ObjectFactory` knows how to create a couple of concrete types. Since it is possible for any string to be supplied to `newObject()`, `ObjectFactory` handles the case when an unknown class is passed. This is where `PropsMap` comes into play. `ObjectFactory` creates a `PropsMap` if it does not recognize the given class name, as we see in Example 16.3. A `PropsMap` is a `DataObject` that holds property values in a hash table, Perl-style.

EXAMPLE 16.3 `src/libs/dataobjects/objectfactory.cpp`

```
[ . . . . ]
DataObject* ObjectFactory::newObject(QString className) {
    DataObject* retval = 0;
    if (className == "UsAddress") {
        retval = newAddress(Country::USA);
    } else if (className == "CanadaAddress") {
        retval = newAddress(Country::Canada);
    } else {
        qDebug() << QString("Generic PropsMap created for new %1")
            .arg(className);
        retval = new PropsMap(className);
        retval->setParent(this); ❶
    }
    return retval;
}
```

❶ Initially set the parent of the new object to the factory.

`newObject()` returns the address of an object whose parent is initially set to an `ObjectFactory` singleton.¹ The parent can be changed later but this ensures that, by default, the object gets deleted with the factory if it's not managed by another object. This is an important measure to prevent memory leaks.

16.1.2 Abstract Factories and Libraries

We now discuss two libraries, `libdataobjects` and `libcustomer`, each with its own `ObjectFactory`, as shown in the UML diagram in Figure 16.1.

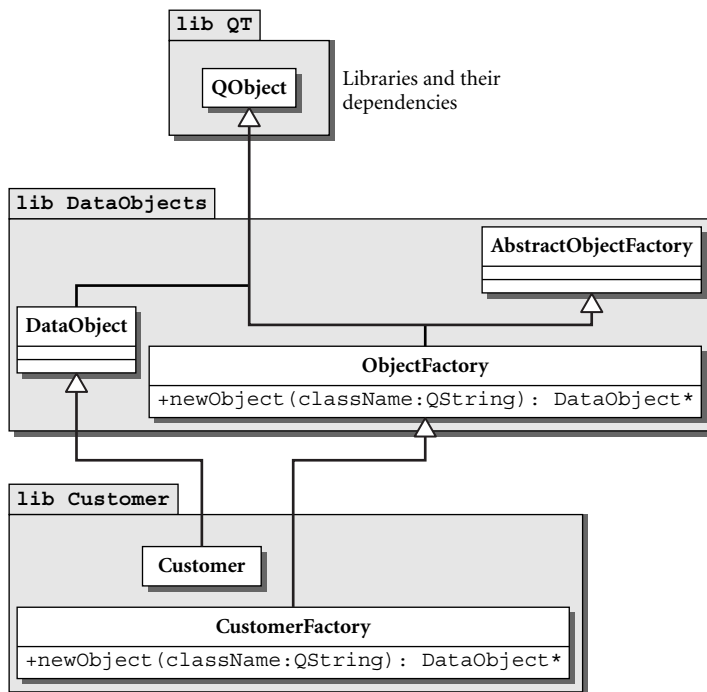


FIGURE 16.1 Libraries and factories

`CustomerFactory`, defined in Example 16.4, *extends* the functionality of `ObjectFactory`.

¹ `newAddress()` also sets the parent to the factory.

EXAMPLE 16.4 `src/libs/customer/customerfactory.h`

```
[ . . . . ]
class CustomerFactory : public ObjectFactory {
    Q_OBJECT
public:
    static CustomerFactory* instance();
    DataObject* newObject(QString classname);
    Customer* newCustomer(QString name = "",
        Country::CountryType country=Country::Undefined);
protected:
    CustomerFactory() {};
};
[ . . . . ]
```

`CustomerFactory` inherits the ability to create `Address` objects from `ObjectFactory`. In addition, it knows how to create `Customer` objects. `CustomerFactory` overrides and extends the `newObject(QString)` method to support more types, as shown in Example 16.5.

EXAMPLE 16.5 `src/libs/customer/customerfactory.cpp`

```
[ . . . . ]
DataObject* CustomerFactory::
newObject (QString className) {
    qDebug() << QString("CustomerFactory::newObject(%1)")
        .arg(className);
    if (className == "Customer")
        return newCustomer();
    if (className == "CustomerList") {
        DataObject* retval = new CustomerList();
        retval->setParent(this); ❶
        return retval;
    }
    return ObjectFactory::newObject(className);
}
```

❶ Guarantee the heap memory is freed eventually when this object factory is destroyed, unless it is reparented and killed sooner.

For each of the three returned objects the parent is set to be the factory. We see this explicitly in the case of a `CustomerList` object. If the `newCustomer()` function is called, it also returns an object that is parented by the factory, as we see in Example 16.6.

EXAMPLE 16.6 `src/libs/customer/customerfactory.cpp`

```
[ . . . . ]

Customer* CustomerFactory::
newCustomer (QString name, Country::CountryType country) {
    Customer* cust = new Customer();
    cust->setName(name);
    cust->setParent(this);
    if (country != 0) {
        Address* defaultAddress = newAddress(country);
        defaultAddress->setParent(cust);
        cust->setAddress(defaultAddress);
    }
    return cust;
}

```

16.1.3 qApp and Singleton Pattern

As we discussed earlier, the Singleton pattern is a specialized factory that is used in situations where at most one instance of a class should be created. We use this pattern to manage a singleton `ObjectFactory` instance. The class definition in Example 16.7 has two distinctive features that characterize this singleton: a non-public constructor and a public static `instance()` function.

EXAMPLE 16.7 `src/libs/dataobjects/objectfactory.h`

```
[ . . . . ]
class ObjectFactory : public QObject, public AbstractFactory {
    Q_OBJECT
public:
    static ObjectFactory* instance() ;
    virtual DataObject* newObject (QString className);
    Address* newAddress (Country::CountryType country =
        Country::Undefined);
    Address* newAddress (QString countryName = "USA");
protected:
    ObjectFactory() {}
};
[ . . . . ]

```

The `instance()` function, defined in Example 16.8, is our singleton factory. It creates an object if needed, but only the first time that the function is invoked. It always returns a pointer to the same object on subsequent calls.

EXAMPLE 16.8 `src/libs/dataobjects/objectfactory.cpp`

```
[ . . . . ]

ObjectFactory* ObjectFactory::instance() { ❶
    static ObjectFactory* singleton = 0;
    if (singleton == 0) {
        singleton = new ObjectFactory();
        singleton->setParent(qApp); ❷
    }
    return singleton;
}
```

❶ static

❷ guarantees this object is deleted when the application is done

For the parent of this object, we set it to `qApp`. This is a pointer to a singleton instance of a `QApplication`, which was presumably created in `main()`. The `QApplication` instance exists precisely as long as the application is running.



WHY NOT USE STATIC FOR OUR SINGLETON? If you make the singleton `ObjectFactory` a static object instead of a heap object, then children of the factory object will be destroyed *after* the `QApplication` is destroyed. Unless there is a compelling reason to the contrary, *an application should not do anything after the `QApplication` has been destroyed*—including object cleanup. Static objects from different files of a multi-file application are destroyed in a linker-dependent order. That order of destruction may cause unintended side-effects (e.g., segmentation faults at termination).

To avoid this problem, you should follow this rule: Each `QObject` should either be allocated on the stack or be allocated on the heap with another `QObject` (or `qApp`) as its parent. This guarantees that there will not be any leftover pieces to be destroyed after the `QApplication` is destroyed.

If a `QObject` is allocated on the heap, and its parent is set to `qApp`, then it is deleted at the “last possible moment.” Children of this singleton factory will be deleted just before the factory is deleted.

16.1.4 Creation Rules and friend Functions (What Friends Are Really For)

You can design a class to have a creation rule, such as: *All new Customers must be created indirectly through a CustomerFactory.* You can enforce this rule by defining only non-public constructors. In particular, make sure that you make the

copy constructor and the assignment operator non-public. If you omit them from the class definition, the compiler will supply *public* versions of these two member functions, thus producing two “loopholes” in your creation rule. For example, this might permit clients to create `Customer` instances that are not children of your factory (which could lead to memory leaks). It is important to document this rule clearly in your class definition, as we did with the comment in Example 16.9.

EXAMPLE 16.9 `src/libs/customer/customer.h`

```
[ . . . . ]
class Customer : public DataObject {
    Q_OBJECT
    [ . . . . ]

    friend class CustomerFactory;
protected:
    Customer(QString name=QString()) { ❶
        setObjectName(name);
    }

    Customer(QString name, QString id, CustomerType type);
```

❶ We declared the constructors of `Customer` as protected, so all `Customer` objects must be created indirectly through a `CustomerFactory`.

In Example 16.9, the constructors are protected.² `CustomerFactory` is declared to be a friend class inside `Customer`. This gives `CustomerFactory` permission to access the non-public members of `Customer`. In this way, we have made it impossible for client code to create `Customer` objects except through the `CustomerFactory`. In Example 16.10, we have a similar setup with the various `Address` classes. The base class, `Address`, is abstract.

EXAMPLE 16.10 `src/libs/dataobjects/address.h`

```
[ . . . . ]
class Address : public ConstrainedDataObject {
    Q_OBJECT
public:
    [ . . . . ]

protected:
    Address(QString addressName = QString()) { ❶
        setObjectName(addressName);
    }
```

continued

² This permits us to write derived classes that reuse this constructor.

```

public:
    virtual Country::CountryType getCountry() = 0;
[ . . . . ]

private:
    QString m_Line1, m_Line2, m_City, m_Phone;
};

```

❶ protected constructor

In the derived classes, defined in Example 16.11, we have protected the constructors and given friend status to a factory, this time `DataObjects::ObjectFactory`. This gives `ObjectFactory` permission to access the non-public members of `UsAddress` and `CanadaAddress`.

EXAMPLE 16.11 `src/libs/dataobjects/address.h`

```

[ . . . . ]

class UsAddress : public Address {
    Q_OBJECT
public:
    Q_PROPERTY( QString State READ getState WRITE setState );
    Q_PROPERTY( QString Zip READ getZip WRITE setZip );
    friend class ObjectFactory; ❶
protected:
    UsAddress( QString name=QString() ) : Address(name) {}
    static QString getPhoneFormat();
public:
    static void initConstraints() ;
[ . . . . ]

private:
    QString m_State, m_Zip;
};

class CanadaAddress : public Address {
    Q_OBJECT
public:
    Q_PROPERTY( QString Province READ getProvince
                WRITE setProvince );
    Q_PROPERTY( QString PostalCode READ getPostalCode
                WRITE setPostalCode );
    friend class ObjectFactory; ❷
protected:
    CanadaAddress( QString name=QString() ): Address(name) {}
    static QString getPhoneFormat();
public:
    static void initConstraints() ;
[ . . . . ]

```

```
private:
    QString m_Province, m_PostalCode;
};
```

- ❶ All new `UsAddress` objects must be created indirectly through an `ObjectFactory`.
- ❷ All new `CanadaAddress` objects must be created indirectly through an `ObjectFactory`.

It is now impossible to create `Address` objects except through the factory or from the derived classes.



If you are writing a multi-threaded application that uses an `ObjectFactory`, you need to be careful about the ownership of objects.

Making a `QObject` the parent of a child in another thread will result in an error message such as the following one:

```
New parent must be in the same thread as the previous parent", file
kernel/qobject.cpp, line 1681
Aborted
```

To make our `ObjectFactory` work in a multi-threaded environment, we can:

1. Explicitly pass the parent when we create each object (to prevent the default parenting behavior)
2. Write the factory's `singleton()` function to return an object that belongs to the local thread

16.1.5 Benefits of Using Factories

One of the benefits of factory patterns is that we can ensure that the created objects are destroyed by assigning them parents before returning them. Any object created by `ObjectFactory` is deleted when the `ObjectFactory` singleton is destroyed (unless the object's parent is changed).

As processing continues, we normally expect the created objects to acquire more appropriate parents. In fact, we can regard created objects that retain the factory parent as “unclaimed” objects. We can clean up all of the heap memory used by unclaimed objects by iterating through the children of the factory and deleting them.³

³ This is similar to the way that garbage-collected heaps work.

Indirect object creation also makes it possible to decide at runtime which class objects to create. This allows the “plugging in” of replacement classes without requiring changes in the client code. In Section 16.2, we will see an example of a method that makes use of factory objects to create trees of connected, client-defined objects based on the contents of an XML file.

Libraries and Plugins

When constructing a large system we group classes together in libraries when they share some common features or need to be used together. Substantial applications generally make use of components from several libraries, some supplied by the development team and some supplied by third-party developers (e.g., Qt from Trolltech). Only the public interface of a library class appears in the client code of reusers. Library designers should be able to change the implementation of any class without breaking client code. Many libraries permit the “plugging in” of outside classes by publishing interfaces and documenting how to implement them. Libraries can facilitate the creation of such plug-in classes by providing a factory base class from which specialized factory classes can be derived as needed.

Another benefit of the factory method (or indirect object creation in general) is that you can enforce post-constructor initialization of objects, including the invocation of virtual functions.

Polymorphism from Constructors

An object is not considered “fully constructed” until the constructor has finished executing. An object’s *vpointer* does not point to the correct vtable until the end of the constructor’s execution. Therefore, calls to methods of `this` from the constructor cannot use polymorphism!

Factory methods are required when any polymorphic behavior is needed during object initialization. Example 16.12 demonstrates this problem.

EXAMPLE 16.12 src/ctorpoly/ctorpoly.cpp

```
#include <iostream>
using namespace std;

class A {
public:
    A() {
        cout << "in A ctor" << endl;
        foo();
    }
    virtual void foo() {
        cout << "A's foo()" << endl;
    }
};

class B: public A {
public:
    B() {
        cout << "in B ctor" << endl;
    }
    void foo() {
        cout << "B's foo()" << endl;
    }
};

class C: public B {
public:
    C() {
        cout << "in C ctor" << endl;
    }

    void foo() {
        cout << "C's foo()" << endl;
    }
};

int main() {
    C* cptr = new C;
    cout << "After construction is complete:" << endl;
    cptr->foo();
    return 0;
}
```

Its output is given in Example 16.13.

EXAMPLE 16.13 src/ctorpoly/ctorpoly-output.txt

```
src/ctorpoly> ./a.out
in A ctor
A's foo()
in B ctor
in C ctor
After construction is complete:
C's foo()
src/ctorpoly>
```

Notice that the wrong version of `foo()` was called while the new `C` object was being constructed. You can find more discussion on vtables in Section 23.1.

EXERCISES: CREATIONAL PATTERNS

1. Complete the implementation of the `Address`, `Customer`, and `CustomerList` classes.

Apply the ideas that we discussed in Section 10.6 to write a `CustomerWriter` class. Make sure that you use the `Q_PROPERTY` features of `Customer` and `Address` so that your `CustomerWriter` class will not need to be rewritten if you change the implementation of `Customer`.

Keep in mind that `Address` objects are stored as children of `Customer` objects. Here is one output format that you might consider using:

```
Customer {
  Id=83438
  DateEstablished=2004-02-01
  Type=Corporate
  objectName=Bilbo Baggins
  UsAddress {
    Line1=52 Shire Road
    Line2=Suite 6
    City=Brighton
    Phone=1234567890
    State=MA
    Zip=02201
    addressName=home
  }
}
```

Another possibility arises if you make use of the `Dataobject::toString()` function.

2. Write a `CustomerReader` class that creates all of its new objects by reusing the `CustomerFactory` class that we supplied.

Write a `CustomerListWriter` and a `CustomerListReader` class that serialize and deserialize lists of `Customer` objects. How much of the `CustomerWriter/Reader` code can you reuse here?

Write client code to test your classes.

16.2 Serializer Pattern Revisited

In this section, we combine `QMetaObjects` with the SAX2 parser to show how one can write a general-purpose XML encoding/decoding tool that works on `QObject`s with well-defined `Q_PROPERTY`s and children. This gives us a nice example that combines the `MetaObject` pattern with the `Serializer` pattern.

To encode and decode `DataObjects` as XML, we must define a mapping scheme. Such a mapping must capture not only the `QObject`'s properties, types, and values, but it must also capture existing relationships between the object and its children, between each child and all of its children, and so on.

The parent-child relationships of XML elements naturally map to `QObject` parents and children. These relationships define a tree structure.

Consider the class definition for `Customer` shown in Example 16.14.

EXAMPLE 16.14 `src/xml/propchildren/customer.h`

```
[ . . . . ]
class Customer : public QObject {
    Q_OBJECT
public:
    Q_PROPERTY( QString Name READ objectName WRITE setObjectName );
    Q_PROPERTY( QDate Date READ getDate WRITE setDate );
    Q_PROPERTY( int LuckyNumber READ getLuckyNumber
                WRITE setLuckyNumber );
    Q_PROPERTY( QString State READ getState WRITE setState );
    Q_PROPERTY( QString Zip READ getZip WRITE setZip );
    Q_PROPERTY( QString FavoriteFood READ getFavoriteFood
                WRITE setFavoriteFood );
    Q_PROPERTY( QString FavoriteDrink READ getFavoriteDrink
                WRITE setFavoriteDrink);

    // typical setters and getters
[ . . . . ]
private:
    QString m_Name, m_State, m_Zip;
    QString m_FavoriteFood, m_FavoriteDrink;
    QDate m_Date;
    int m_LuckyNumber;
};
[ . . . . ]
```

Exploiting the ability of `QObject` subclasses to maintain a collection of child objects, we define a `CustomerList` class in Example 16.15 that stores `Customers` as children.

EXAMPLE 16.15 `src/xml/propchildren/customerlist.h`

```
#ifndef CUSTOMERLIST_H
#define CUSTOMERLIST_H

#include <QList>
#include "customer.h"

class CustomerList : public QObject {
    Q_OBJECT
public:
    CustomerList(QString listname = QString()) {
        setObjectName(listname);
    }
    QList<Customer*> getCustomers();
    static CustomerList* sample();
};

#endif
```

An example of the desired XML format for storing the data of a `CustomerList` is shown in Example 16.16.

EXAMPLE 16.16 `src/xml/propchildren/customerlist.xml`

```
<object class="CustomerList" name="Customers" >

  <object class="Customer" name="Simon" >
    <property name="Name" type="QString" value="Simon" />
    <property name="Date" type="QDate" value="1963-11-22" />
    <property name="LuckyNumber" type="int" value="834" />
    <property name="State" type="QString" value="WA" />
    <property name="Zip" type="QString" value="12345" />
    <property name="FavoriteFood" type="QString" value="Donuts" />
    <property name="FavoriteDrink" type="QString" value="YooHoo" />
  </object>

  <object class="Customer" name="Raja" >
    <property name="Name" type="QString" value="Raja" />
    <property name="Date" type="QDate" value="1969-06-15" />
    <property name="LuckyNumber" type="int" value="62" />
    <property name="State" type="QString" value="AZ" />
    <property name="Zip" type="QString" value="54321" />
  </object>
</object>
```

```

    <property name="FavoriteFood" type="QString" value="Mushrooms" />
    <property name="FavoriteDrink" type="QString" value="Jolt" />
  </object>
</object>

```

With this kind of information in an input file, we should be able to fully reconstruct not only the properties and their types, but also the tree structure of parent-child relationships between objects for a `CustomerList`.

16.2.1 Exporting to XML

We define in Example 16.17 a simplified class that can be used to export the current state of a `QObject` to an XML string with elements that contain, for each property, its name, type, and value.

EXAMPLE 16.17 `src/xml/propchildren/xmlexport.h`

```

[ . . . . ]
class XMLExport {
public:
    virtual ~XMLExport() {}
    virtual QString objectToXml(const QObject* ptr,
                               int indentLevel=0);
};
[ . . . . ]

```

In Example 16.18 we show the definition of `objectToXml()`, a recursive function that constructs strings for each of the object's properties and then iterates over the object's children, recursively calling `objectToXml()` on each child.

EXAMPLE 16.18 `src/xml/propchildren/xmlexport.cpp`

```

[ . . . . ]
QString XMLExport::objectToXml(const QObject* doptr,
                               int indentLevel) {

    QStringList result;
    QString indentspace;

    indentspace.fill(' ', indentLevel*3);
    const QMetaObject* meta = doptr->metaObject();
    result += QString("\n%1<object class=\"%2\" name=\"%3\" >").
        arg(indentspace).
        arg(meta->className()).
        arg(doptr->objectName());
}

```

continued

```

for (int i= 0; i < meta->propertyCount(); ++i) {
    QMetaProperty qmp = meta->property(i);
    const char* propName = qmp.name();
    if (strcmp(propName, "objectName")==0)
        continue;
    QVariant qv;
    if (qmp.isEnumType()) {
        QMetaEnum qme = qmp.enumerator();
        qv = qme.valueToKey(qv.toInt());
    } else {
        qv = doptr->property(propName);
    }

    result += QString (
        "%1 <property name=\"%2\" type=\"%3\" value=\"%4\" />"
        ).arg(indentSpace).arg(propName). arg(qv.typeName())
        .arg(variantToString(qv));
}

QObjectList childlist = doptr->findChildren<QObject*> (QString());
foreach (QObject* objptr, childlist) {
    if (objptr->parent()==doptr) {
        result += objectToXml(objptr, indentLevel+1);
    }
}
result += QString("%1</object>\n").arg(indentSpace);
return result.join("\n");
}
[ . . . . ]

```

- ❶ Iterate through each property.
- ❷ Iterate through the child list.
- ❸ findChildren also includes grandchildren and great-great grandchildren, so we skip over those.
- ❹ recursive call

objectToXml() uses Qt's properties and QMetaObject facilities to reflect on the class. As it iterates it appends each line to a QStringList. When iteration is complete, the <object> is closed. The return QString is then produced quickly by calling QStringList::join("\n").

16.2.2 Importing Objects with an Abstract Factory



Section 14.2

The importing routine is a bit more sophisticated than the exporting routine, and it has a couple of interesting features.

- It parses XML using the SAX parser.
- Depending on the input, it creates objects.
- The number and types of objects, as well as their parent-child relationships, must be reconstructed from the information in the file.

Example 16.19 shows the class definition for `DataObjectReader`.

EXAMPLE 16.19 `src/libs/dataobjects/dataobjectreader.h`

```
[ . . . . ]
#include <QString>
#include <QStack>
#include <QQueue>
#include <QXmlDefaultHandler>

class AbstractFactory;
class DataObject;
class DataObjectReader : public QXmlDefaultHandler {
public:
    DataObjectReader (AbstractFactory* factory=0) :
        m_Factory(factory), m_Current(0) { }
    DataObjectReader (QString filename,
        AbstractFactory* factory=0);

    void parse(QString text);
    void parseFile(QString filename);
    DataObject* getRoot();
    ~DataObjectReader();

    // callback methods from QXmlDefaultHandler
    bool startElement( const QString & namespaceURI,
        const QString & name,
        const QString & qualifiedName,
        const QXmlAttributes & attributes );
    bool endElement( const QString & namespaceURI,
        const QString & localName,
        const QString & qualifiedName);
    bool endDocument();
private:
    void addCurrentToQueue();
    AbstractFactory* m_Factory;
    DataObject* m_Current;
    QQueue<DataObject*> m_ObjectList;
    QStack<DataObject*> m_ParentStack;
};
[ . . . . ]
```

Figure 16.2 shows the relationships between the various classes that we will be using.

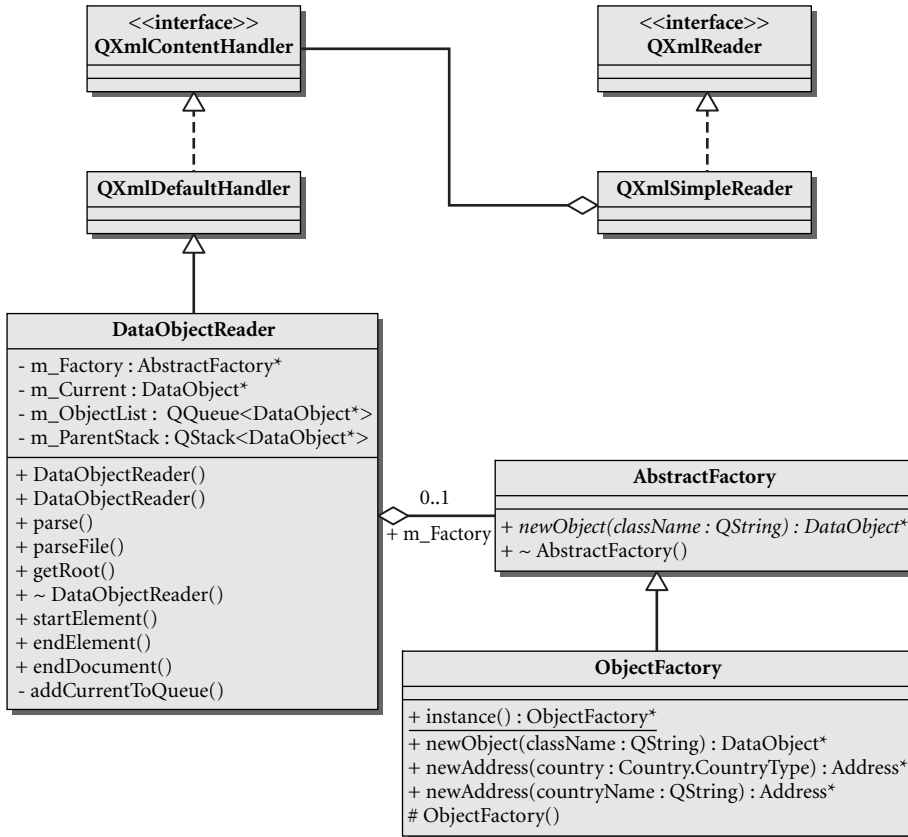


FIGURE 16.2 DataObjectReader and its related classes

DataObjectReader is derived from QXmlDefaultHandler, which is a plugin for the QXmlSimpleReader. AbstractFactory is a plugin for DataObjectReader. When we create a DataObjectReader, we must supply it with a concrete class, such as ObjectFactory or DataObjectFactory.

DataObjectReader is now completely separate from the specific types of objects that it can create. To use it with your own types, just derive a factory from AbstractFactory for them.

Think about Example 16.16 as you read the code that constructs objects from it in the code that follows.

`startElement()` is called when the SAX parser encounters the initial tag of an XML element. As we see in Example 16.20, the parameters to this function contain all the information we need to create an object. All other objects that are encountered between `startElement()` and the matching `endElement()` are children of `m_Current`.

EXAMPLE 16.20 src/libs/dataobjects/dataobjectreader.cpp

```
[ . . . . ]

bool DataObjectReader::startElement( const QString &,
                                     const QString & elementName,
                                     const QString &,
                                     const QDomAttributes & atts) { ❶
    if (elementName == "object") {
        if (m_Current != 0) ❷
            m_ParentStack.push(m_Current); ❸
        QString classname = atts.value("class");
        QString instancename = atts.value("name");
        if (m_Factory == 0) {
            m_Current =
                ObjectFactory::instance()->newObject(classname);
        } else {
            m_Current=m_Factory->newObject(classname);
        }
        m_Current->setObjectName(instancename);
        if (!m_ParentStack.empty()) { ❹
            m_Current->setParent(m_ParentStack.top());
        }
        return true;
    }
    if (elementName == "property") {
        QString fieldType = atts.value("type");
        QString fieldName = atts.value("name");
        QString fieldValue = atts.value("value");
        QVariant qv = variantFrom(fieldType, fieldValue);
        bool ok = m_Current->setProperty(fieldName, qv);
        if (!ok) {
            qDebug() << "setProperty(" << fieldName << ") failed";
        }
    }
    return true;
}

```

- ❶ Unnamed parameters are a way of avoiding “parameter not used” warnings from the compiler. It is necessary to include the parameters, even though we do not need them for this application, so that the signature matches that of the base class method and polymorphic overrides will be properly called.
- ❷ if we are already inside an <object>
- ❸ Keep track of the current parent.
- ❹ If this element has a parent, it is on the top of the stack. Set its parent.

The Object is “finished” when we reach `endElement()`, which is defined in Example 16.21.

EXAMPLE 16.21 `src/libs/dataobjects/dataobjectreader.cpp`

```
[ . . . . ]

bool DataObjectReader::endElement( const QString & ,
                                   const QString & elementName,
                                   const QString & ) {
    if (elementName == "object") {
        if (!m_ParentStack.empty())
            m_Current = m_ParentStack.pop();
        else {
            addCurrentToQueue();
        }
    }
    return true;
}
```

`DataObjectReader` uses an Abstract Factory to do the actual object creation.

The callback function, `newObject(QString className)`, creates an object that can hold all of the properties described in `className`. `ObjectFactory` creates “pseudo-objects” that are not exactly the `CustomerList` and `Customer` classes, but they “mimic” them well enough that the export/import process works round-trip. You can write a concrete factory that returns the proper types for each `classname` if you want the de-serialized tree to have the same types as the objects in the original tree.

Each time a new address type is added to this library, we can add another `else` clause to the `createObject` function, as shown in Example 16.22.

EXAMPLE 16.22 `src/libs/dataobjects/objectfactory.cpp`

```
[ . . . . ]

DataObject* ObjectFactory::newObject(QString className) {
    DataObject* retval = 0;
    if (className == "UsAddress") {
        retval = newAddress(Country::USA);
    } else if (className == "CanadaAddress") {
        retval = newAddress(Country::Canada);
    } else {
        qDebug() << QString("Generic PropsMap created for new %1 ").
            arg(className);
        retval = new PropsMap(className);
        retval->setParent(this); ❶
    }
    return retval;
}
```

❶ Initially set the parent of the new object to the factory.

16.3 The Façade Pattern

A class that uses the Façade pattern provides “a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier [and safer] to use.”⁴ When a class interface is too complicated to use effectively (leads to hard-to-debug errors from improper use) or does not use a programming style that fits into our larger framework, a façade should be used. A façade is a class with a clear simple interface that encapsulates and hides a complicated set of classes and/or functions.

Experience gained from the struggle to reuse classes with difficult interfaces can provide valuable motivation for designing elegant, friendly, and useful interfaces for your own classes.

MP3 files have become a popular format for storing songs and other audio content. Each MP3 file stores audio data which, when processed by a program such as XMMS, can be output as audio. The file also stores **metadata**, or structured information about the audio content. The defacto standard for specifying MP3 metadata is called ID3.

Using `id3lib`, an open-source library, the metadata can be loaded into an object of class `ID3_Tag`. An `ID3_Tag` is a collection of `ID3_Frame` objects plus file data. An `ID3_Frame` object consists of a collection of `ID3_Field` objects. An `ID3_Field` can represent three possible types of data: integers, binary data, or text strings. Figure 16.3 shows the relationships between the types in a collaboration diagram originally generated by Doxygen.

`id3lib`⁵ is an old, fast, open-source C/C++ library for reading and writing ID3v2 tags from mp3 files. Because of its arcane programming style, it is being shunned by KDE developers in favor of a newer library, `taglib`,⁶ which is more flexible (can handle other file formats as well as MP3) and uses a more modern programming style.

⁴ [Gamma 95]

⁵ <http://id3lib.sourceforge.net/>

⁶ <http://developer.kde.org/~wheeler/taglib.html>

EXAMPLE 16.23 `src/facade/id3lib-usage.cpp`

```
[ . . . . ]
#include <id3/tag.h>
#include <id3/misc_support.h>
#include <QString>

QString getStringField(ID3_Tag &tag, ID3_FrameID id,
    ID3_FieldID fieldid) {

    static char buffer[256];
    const int size=255;

    ID3_Frame* myFrame = tag.Find(id);
    if (myFrame) {
        ID3_Field *myField = myFrame->GetField(fieldid);
        if (myField)
            myField->Get(buffer, size);
    }
    return QString(buffer);
}

QString getPreference(QString filename) { ❶
    QString retval;

    ID3_Tag tag;
    tag.Link(filename.toAscii());

    QString commentType = getStringField(tag, ID3FID_COMMENT,
        ID3FN_DESCRIPTION);
    if (commentType == "MusicMatch_Preference") {
        retval = getStringField(tag, ID3FID_COMMENT, ID3FN_TEXT);
        return retval;
    } else
        return "Undefined";
}
[ . . . . ]
```

❶ gets the MusicMatch preference

The code in Example 16.23 has some style issues that make it hard to fit into our Qt-style framework.

- First, char arrays should be avoided whenever possible. We use `QString` instead, to support Unicode, improve readability, and reduce the chance of bugs.
- Client code should not need to deal with `ID3Frame` or `ID3Field` objects, or their ID enumerators.
- Class and function capitalization rules of `id3lib` are inconsistent with our style guidelines (see Section 3.4).

16.3.1 Functional Façade

`id3lib` has a collection of convenience functions for getting and setting the more popular tags without needing to use the `enum` constants or to deal with frames and fields. The convenience functions are defined in `misc_support.cpp`.⁸ In a sense, these functions are a façade for the `ID3_tag` class and related functions. Unfortunately, the convenience functions are ordinary C functions, which means they use C-style coding to access ID3 data.

Using these functions instead of the `ID3_tag` class and functions, client code does become simpler. For example, to get or add the genre of a song, we can use these functions:

```
char * ID3_GetGenre ( const ID3_Tag * tag);
ID3_Frame * ID3_AddGenre (ID3_Tag * tag, size_t genre, bool replace);
```

The `getPreference()` function, rewritten to use this library, is also much simpler:

```
QString getPreference(QString filename) const {
    ID3_Tag tag;
    tag.Link(filename);
    return ID3_GetComment(&tag, "MusicMatch_Preference");
}
```

So, the advantages to using these functions are

- They have already been tested and debugged.
- They have removed the need to use `enum` constants, frames, or fields.

The disadvantages to using this library directly in C++ client code are

- We have an object that maintains the state of an MP3 tag record (`ID3_Tag`), but we are no longer using its interface to access and change its values.
- We are using C coding style in client code.

16.3.2 Smart Pointers: `auto_ptr`

A **wrapper** encapsulates and manages at least one other object. If that object is a heap object, we can use the Standard Library `auto_ptr` to ensure that the object is always destroyed when the wrapper is.

`auto_ptr` is a **template type**, so it can be instantiated for use with any other type. In Example 16.24, we work with the `Customer` type.

⁸ http://id3lib.sourceforge.net/api/misc_support.cpp.html

EXAMPLE 16.24 auto_ptr code fragment

```

for (int i=0; i< someNumber; ++i) {
    auto_ptr<Customer> custPtr;
    auto_ptr<Customer> custPtr2 (new Customer());
    custPtr = custPtr2;
}

```

- ❶ The loop is here to demonstrate how a block of code creates and destroys its local objects on the stack.
- ❷ a null `auto_ptr`
- ❸ `custPtr2` is initialized to point to and manage the new `Customer` object.
- ❹ `custPtr` takes ownership of `custPtr2`, meaning that `custPtr2` will be `NULL` after this statement is over.
- ❺ At the end of each iteration, the local `custPtr` goes out of scope, and the heap `Customer` instance gets destroyed.

An `auto_ptr<Customer>` is *type-restricted*, to point to objects derived from `Customer`, as specified in the template parameter. An attempt to use this to point to any other type will result in a compiler error.

When an `auto_ptr` is destroyed, the pointed-to heap object is deleted. Therefore, when the above code in Example 16.24 is executed, there will be no memory leaks, even though the loop created many new heap objects and the code contains no corresponding `delete`.

Assignment with `auto_ptr` is very different from assignment with other types. Usually, the right side is not changed. However, the `auto_ptr` on the right side of an assignment always becomes `NULL`, while the left side takes ownership of the pointed-to object. This guarantees that only one `auto_ptr` is ever pointing to an object and ensures that the object will get deleted exactly once by its `auto_ptr`.

Guarded pointers⁹ such as the `auto_ptr` are frequently used in wrappers and façades, as we show in the next example.

16.3.3 FileTagger: Façade Example

In this section, we discuss a Qt-style façade for `id3lib`. To this end, we employ another pattern that is closely related to the façade.

⁹ `QPointer` is another guarded pointer, similar to `auto_ptr`, but specifically for use on `QObject`s. Guarded pointers are discussed in more detail in Section 19.9.

The **Adaptor pattern** converts the interface of a class into another interface that clients expect, so classes that originally had incompatible interfaces can be used together. Adaptor is also known as the **Wrapper pattern**.¹⁰

Figure 16.4 shows the relationships between the classes that we will use for `FileTagger`, a façade for `id3lib`, and a wrapper for an `ID3_Tag`.

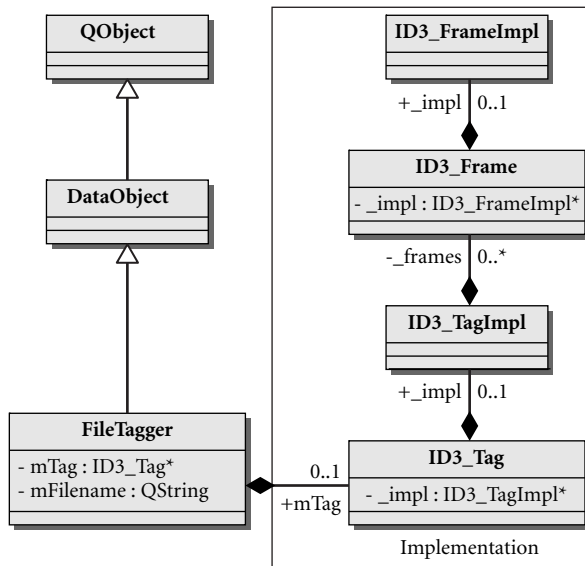


FIGURE 16.4 `FileTagger` and the classes it fronts for

Here are the requirements for `FileTagger`.

1. It must safely allocate and free any resources that it uses (such as `ID3_Tag` objects from the heap).
2. It should provide a `QObject` properties interface compatible with `moc`.
3. It must expose all features of `id3lib` that we need to use, so that it is not necessary to include the `id3lib` headers from any other source code module.
4. It must provide a clean and simple interface with self-explanatory function names.

¹⁰ When the same pattern is arrived at by different people, and given different names, its importance is emphasized.

5. It must have getters and setters for the standard tag items (artist, album, title, genre, preference, track number) as specified in an abstract class, `Mp3Song`.
6. It must hide all of `id3lib`'s enumerated constants and noncompliant capitalization conventions from the users of this class.
7. It must not use C-style `char` arrays to represent strings. Instead, it should use Unicode `QStrings`.
8. It should emit signals when its properties are changed.
9. It should define slots for setting properties.

Example 16.25 shows the class definition for the `FileTagger` class.

EXAMPLE 16.25 `src/libs/filetagger/filetagger.h`

```
[ . . . . ]

#include <QString>
#include <memory>
#include <dataobject.h>
#include "mp3song.h"
#include <id3/tag.h>

class FileTagger : public DataObject, public Mp3Song {
    Q_OBJECT
public:
    Q_PROPERTY( QString Artist READ getArtist
               WRITE setArtist );
    Q_PROPERTY( QString TrackTitle READ getTrackTitle
               WRITE setTrackTitle );
    Q_PROPERTY( QString AlbumTitle READ getAlbumTitle
               WRITE setAlbumTitle );
    Q_PROPERTY( QString TrackNumber READ getTrackNumber
               WRITE setTrackNumber );
    Q_PROPERTY( QString Genre READ getGenre
               WRITE setGenre);
    Q_PROPERTY( QString Comment READ getComment
               WRITE setComment);
    Q_PROPERTY( QString Preference READ getPreference
               WRITE setPreference);
    Q_PROPERTY( QString Filename READ getFilename
               WRITE setFilename );
    // read only properties
    Q_PROPERTY( QString Url READ getUrl);
    Q_PROPERTY( int TrackTime READ getTrackTime);
public:
    FileTagger( QString filename = "" );
    ~FileTagger();
    bool isValid() const;
```

continued

```

    /* getters ...*/
    QString getFilename() const {return m_Filename; }
    QString getPreference() const;
[ . . . . ]

private:
    std::auto_ptr<ID3_Tag> m_Tag;
    QString m_Filename;
};

```

In Example 16.26 we show the implementation details for some of the member functions. The setter for `m_FileName` not only assigns a value to that member but it also clears the `ID3_Tag` and establishes a link between that tag and the named file.

`auto_ptr` works as a smart pointer, managing the memory for us so that the destructor does not need to delete that memory.

EXAMPLE 16.26 `src/libs/filetagger/filetagger.cpp`

```

[ . . . . ]

// macro for converting QSrtings to ASCII
#define ASCII toAscii().data()

bool FileTagger::isValid() const {
    if (m_Tag.get() == NULL)           ❶
        return false;
    return m_Tag->HasV1Tag ();         ❷
}

void FileTagger::setFilename(const QString & filename) {

    std::auto_ptr<ID3_Tag> newTag( new ID3_Tag() );
    m_Tag = newTag;                    ❸
    m_Filename = filename;
    m_Tag->Link(m_Filename.ASCII);     ❹
    emit propertyChanged("all", "all");
}

FileTagger::~FileTagger() {
    // delete m_Tag;                    ❺
}

```

- ❶ `get()` returns the managed pointer.
 - ❷ `auto_ptr` has an operator `>` that lets us use it like a pointer.
 - ❸ We can't assign an `auto_ptr` to anything but another `auto_ptr`.
 - ❹ using ASCII conversion macro
 - ❺ not needed with `auto_ptr`
-

The other setters and getters, shown in Example 16.27, make use of the C-style convenience functions from the ID3 misc_support library. Those functions have names like `ID3_AddSomething()` or `ID3_GetSomething()`, and each takes a

regular pointer to an `ID3_Tag`. The `auto_ptr` has a `get()` function for returning that pointer.

EXAMPLE 16.27 `src/libs/filetagger/filetagger.cpp`

```
[ . . . . ]

void FileTagger::setPreference(const QString& prefstr) {
    ID3_AddComment(m_Tag.get(), prefstr.ASCII,
                  "MusicMatch_Preference", "en", true);
}

QString FileTagger::getPreference() const {
    return ID3_GetComment(m_Tag.get(), "MusicMatch_Preference");
}

void FileTagger::setGenre(const QString& newGenre) {
    ID3_AddGenre(m_Tag.get(), newGenre.ASCII, true);
}

QString FileTagger::getGenre() const {
    return ID3_GetGenre(m_Tag.get());
}

```

Each of the ID3 fields we plan to use is now mapped to a Qt property with a proper getter and a setter. No `char*` are needed to work with ID3 tags for any application that uses this class.

EXERCISES: THE FAÇADE PATTERN

1. Build view classes for the `Customer` and `CustomerList` classes that you defined in “Exercises: Creational Patterns” in Section 16.1.6.
 - `CustomerView` should display the current values of all `Customer` attributes.
 - `CustomerList` should only display a list of `Customer` names. If the user clicks on a name in that list, the corresponding `CustomerView` should appear.
2. Compile and install `libid3` according to the instructions in Section 25.4. Write a test program to verify that you can read and write ID3 tags.

POINTS OF DEPARTURE

1. Try to come up with another `UserType` you might want to add to `QVariant` for a new type of `InputField`.
2. Further discussion of `moc` and marshalling objects using metaobjects can be found in *Qt Quarterly*.¹¹

¹¹ <http://doc.trolltech.com/qc/qc14-metatypes.html>

REVIEW QUESTIONS

1. How can a creational pattern help manage object destruction?
2. How can properties help us write a more general-purpose `Writer`?
3. How can an Abstract Factory help us write a more general-purpose `Reader`?
4. What is `auto_ptr` used for?
5. What is special about assignment between `auto_ptr` objects?
6. We can create a `FormModel` in a number of ways. One approach is to create `Question` objects directly and add them. Another way is to create a `FormModel` from a `DataObject`. Why would we use one technique instead of the other?
7. Name other examples of façades that we have worked with in the book. Explain why they are façades (or wrappers).


17

CHAPTER 17

Models and Views

The Model-View pattern describes techniques of separating the underlying data (the model) from the class that presents the user with a GUI (the view). In this chapter we will see a model for a form, and a couple of ways to view and enter data into it. Qt model and view classes are discussed, and we will see examples of lists, trees, and tables.

17.1 M-V-C: What about the Controller? ...	392
17.2 Dynamic Form Models	393
17.3 Qt 4 Models and Views	409
17.4 Table Models	411
17.5 Tree Models	417



In several earlier examples we saw code that maintained a clean separation between **model classes** that represent data and **view code** that presents a user interface. There are several important reasons for enforcing this separation.

First of all, separating model from view reduces the complexity of each. Model code and view code have completely different maintenance imperatives—changes are driven by completely different factors—so it is much easier to maintain each of them when they are kept separate. Furthermore, the separation of model from view makes it possible to maintain several different, but consistent, views of the same data model. The number of sophisticated view classes that can be reused with well-designed models is constantly growing.

Some older GUI toolkits offer lists, trees, and tables, but they require the developer to store model data inside them. Storing data inside view classes leads to a strong dependency between the user interface and the underlying data model. This dependency makes it very difficult to reuse the view classes.

17.1 M-V-C: What about the Controller?

There is a third tier to the model/view structure: the controller. **Controller code** is code that manages the interactions among events, models, and views. Factory methods, and creation and destruction code in general fall into the realm of the controller.

Model-View-Controller (MVC), illustrated in Figure 17.1, is a design pattern that is used for applications in which a variety of views of the same data need to be maintained. The pattern specifies that the model code (responsible for maintaining the data), the view code (responsible for displaying all or part of the data in various ways), and the controller code (responsible for handling events that impact the data or the model) be kept as separate as possible from one another. This separation allows views and controllers to be added or removed without requiring changes in the model.

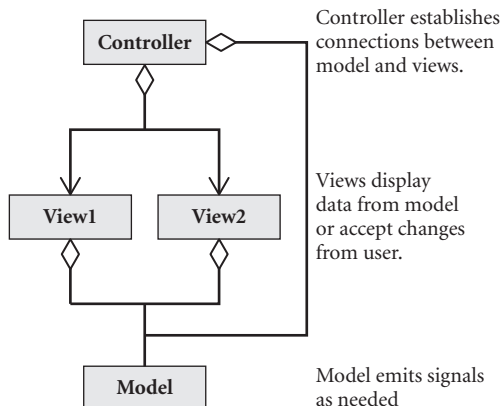


FIGURE 17.1 Model-View-Controller pattern

A **controller class** is a class whose specific purpose is to encapsulate controller code. A complex application might have multiple controllers for the different sub-components, or layers, of the application.

Qt 4 classes that are considered controller classes include `QApplication`, `QAction`, and their derived types. Code that connects signals to slots can also be considered controller code. As we shall see, keeping controller code out of model and view classes will yield additional design benefits.

17.2 Dynamic Form Models

A collection of questions and blanks to fill in the answers is called a **form**. Quite common in GUI applications, a form is used whenever a series of questions needs to be asked of the user. There are many ways to design and implement forms. In this section, we will make a model for the form, and then a view for it.

EXAMPLE 17.1 `src/libs/forms/testform.cpp`

```
[ . . . . ]
class BridgeKeeper : public FormModel {
public:
    BridgeKeeper();
};
```

continued

```

BridgeKeeper::BridgeKeeper() {
    FormFactory ff;
    *this << ff.newQuestion("name", "What is your name?");
    *this << ff.newQuestion("quest", "What is your quest?");
    QStringList colors;
    colors << "red" << "blue" << "green" << "orange";
    *this << ff.newQuestion("color",
        "What is your favorite color?", colors);
    *this << ff.newQuestion("speed",
        "What is the mean air speed of an unladen swallow?",
        QVariant::Int);
}

```

The model created in Example 17.1 represents a form containing questions of different “types,” where the first two are simple string inputs, but the last two are constrained to a set of possible values. The `main` program creates a model and a view, and hooks them together. In Example 17.2, you can think of `main` as the controller.

EXAMPLE 17.2 `src/libs/forms/testform.cpp`

```

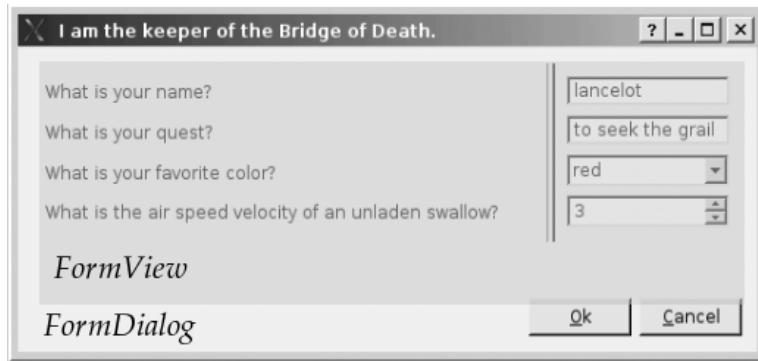
[ . . . . ]

int main(int argc, char** argv) {
    QApplication a(argc, argv);
    QMainWindow mw;
    BridgeKeeper keeper;
    qDebug() << keeper.toString();
    FormDialog fv(&keeper);
    fv.setWindowTitle("I am the keeper of the Bridge of Death.");
    mw.setCentralWidget(&fv);
    mw.setVisible(true);
    int retval = a.exec();
    QVariant speed = keeper.property("speed");
    QVariant color = keeper.property("color");
    QVariant quest = keeper.property("quest");
    QVariant name = keeper.property("name");
    if (color.toString() == "blue") {
        qDebug() << "no, I mean red! aaaaaahhhhhhhhhhh!" ;
    }
    else {
        qDebug() << "My name is " << name.toString()
            << ", and I " << quest.toString()
            << ". My favorite color is " << color.toString()
            << ". The speed is " << speed.toInt();
    }
    return retval;
}

```

In Example 17.2, you can think of `main` as the controller.

The `FormDialog` below is automatically generated from the model above, even though it does not depend on the specific model.



The dialog embeds a `FormView`, which contains the actual input widgets.

1. View classes access `Questions` only through the public polymorphic interface.
2. Model classes emit signals to communicate information to views, instead of invoking objects directly through references or pointers passed around as function parameters.
3. The code that does depend on both model and view (or on specific types of `Question`) is kept separate as controller code.

A `FormModel` wraps a collection of `Questions`, which are model classes. A `FormView` wraps a collection of `InputFields`, which are views (because they derive from `QWidget`), but encapsulate complex input widgets.

In the Doxygen collaboration diagram in Figure 17.2, there is a 1:1 correspondence between `InputField` and `Question`, but the classes are *strictly decoupled*.¹

A `Question` models (ideally) all of the information needed by `FormFactory` to create an appropriate `InputField`. A `FormView` is a grouping of input widgets. An input widget serves as a proxy, or delegate, between Qt input widgets and `Question`-derived models.

¹“Strictly decoupled” means that they know nothing about each other.

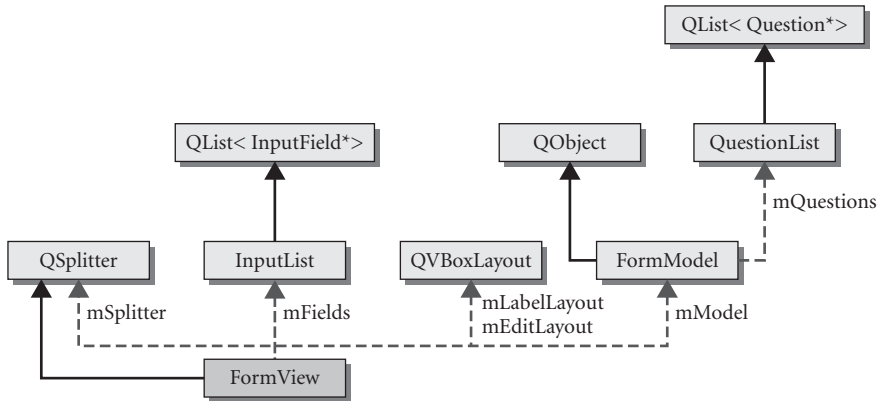


FIGURE 17.2 Forms

The **Strategy pattern** encapsulates each member of a family of algorithms so that they can be selected independently by clients. Each encapsulated algorithm is called a **strategy**.

Figure 17.3 shows that an `InputField` can be a variety of things. Because Qt input widgets² do not have a common `QVariant`-based interface for getting and setting data, the `InputField` serves as an **adaptor**, that provides a property-like interface. `InputField` uses the Strategy pattern to organize the getters and setters for different types as virtual functions.

With a hierarchy of views, we end up with an extensible framework for adding other kinds of `InputFields` later.

EXERCISES: DYNAMIC FORM MODELS

1. Add a `DoubleInputField` class, derived from `InputField`, and update the `FormFactory` to return it as needed.
2. Write a testcase with a `QVariant` `Question` in it, and verify that a `QDoubleSpinBox` shows up.

² `QLineEdit`, `QComboBox`, `QDateEdit`, `QSpinBox`

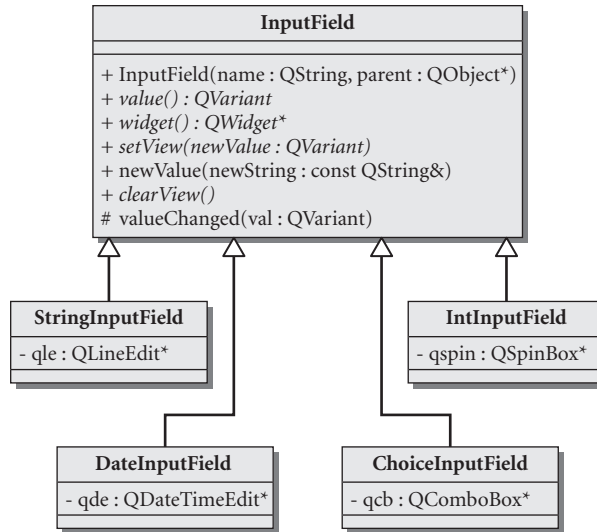


FIGURE 17.3 Input fields

17.2.1 Form Models

In addition to different kinds of input widgets (for the different data types), there can also be different kinds of `Question` models for getting/setting the data in different places.

The `FormModel` and the `Question` classes, shown in Figure 17.4, are two adjoining layers in the model. Because they are models, they are meant to be very simple classes, holding data but containing no GUI or controller code.

`FormModel` provides a simple operation, `setValues()`, for updating all of its `Question`'s values, shown in Example 17.3.

EXAMPLE 17.3 `src/libs/forms/formmodel.cpp`

```

[ . . . . ]

bool FormModel::setValues(QList<QVariant> list) {
    bool retval = true;
    for (int i=0; i<list.size(); ++i) {
        QString str = list.at(i).toString();
        Question* q = m_Questions.at(i);
        retval = q->setValue(str) && retval;
    }
    emit modelChanged();
    return retval;
}

```

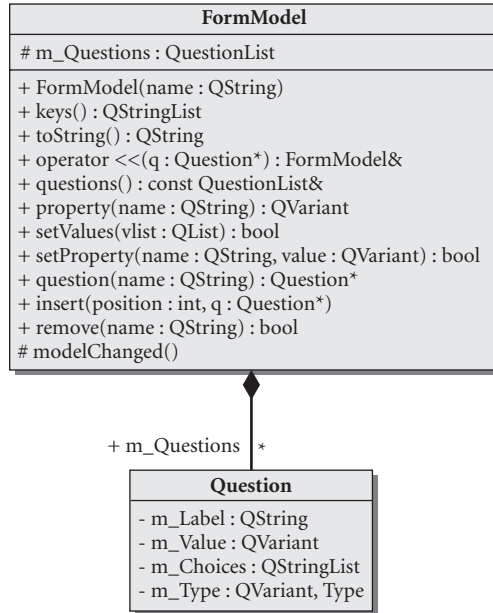


FIGURE 17.4 FormModel and Question

`Question` encapsulates all the things that are needed for an interaction with a user, including the type and value of the expected answer. The constructors, declared in Example 17.4, are `protected` because we will use a factory to create `Question` objects.

EXAMPLE 17.4 src/libs/forms/question.h

```

[ . . . . ]
class Question : public QObject {
    Q_OBJECT
protected:
    Question(QString name, QString label = QString(),
             QVariant::Type type=QVariant::String);
    Question(QString name, QString label,
             QStringList choices, bool open=false);
    Question() {}
public:
    virtual Qt::ItemFlags flags() const;
    virtual QString toString() const;
    virtual QVariant value() const;

```

```

    virtual QVariant::Type type() const ;
    QStringList choices() const ;
    QString label() const {return m_Label;}
    virtual ~Question() {}
public slots:
    virtual bool setValue(QVariant newValue);
signals:
    void valueChanged();
protected:
    void setType(QVariant::Type type) ;
    void setLabel(QString label) ;
private:
    QString m_Label;
    QVariant m_Value;
    QStringList m_Choices;
    QVariant::Type m_Type;
};
[ . . . . ]

```

When we need a `Question` instance, `FormFactory` creates it by using one of the protected constructors, defined in Example 17.5.

EXAMPLE 17.5 `src/libs/forms/question.cpp`

```

[ . . . . ]

Question::Question( QString name, QString label, QVariant::Type t):
    m_Label(label) {
        setObjectName(name);
        if (m_Label == QString())
            m_Label = name;
        m_Value = QVariant(t);
        m_Type = m_Value.type();
    }

Question::Question( QString name, QString label, QStringList choices,
                    bool) : m_Label(label), m_Choices(choices) {
    setObjectName(name);
    if (m_Label == QString())
        m_Label = name;
    m_Type = QVariant::StringList;
}

```

17.2.2 Form Views

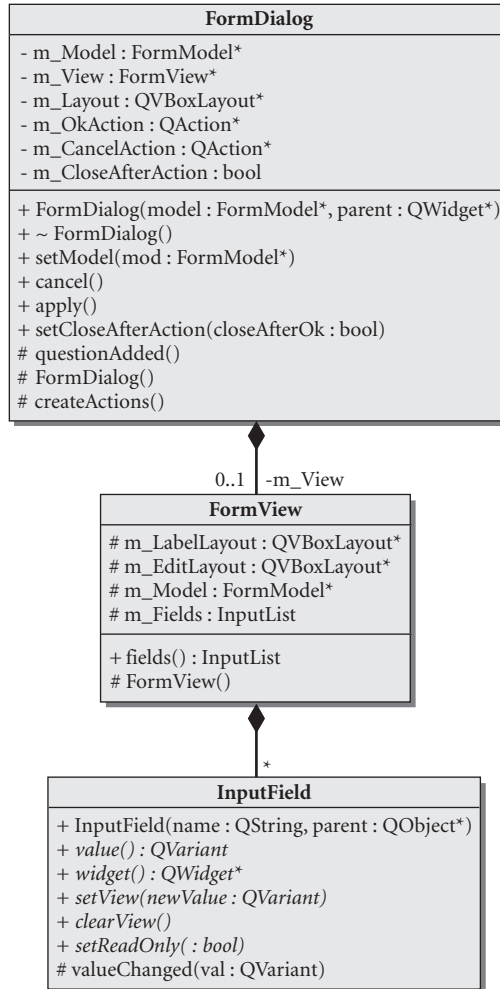


FIGURE 17.5 Form views

The view classes, shown in Figure 17.5, are separated into three layers.

1. `FormDialog` includes the buttons and actions plus some management/controller code.
2. `FormView` is solely responsible for holding layouts for labels and input widgets.
3. `InputField` is responsible for a single input widget.

`FormView` can be automatically created from a `FormModel` without any knowledge of the individual `InputField` or `Question` types. This is thanks to the `createEditor` Factory method, used in Example 17.6, which returns polymorphic objects.

EXAMPLE 17.6 `src/libs/forms/formfactory.cpp`

```
[ . . . . ]

FormView* FormFactory::formView(FormModel* mod) {
    FormView* retval = new FormView();
    retval->m_Model = mod;
    retval->m_LabelLayout = new QVBoxLayout();
    retval->m_EditLayout = new QVBoxLayout();
    foreach (Question* q, mod->questions()) {
        QLabel* label = new QLabel(q->label());
        retval->m_LabelLayout->addWidget(label);
        InputField* editField = createEditor(q); ❶
        retval->m_Fields += editField;
        label->setBuddy(editField->widget());
        retval->m_EditLayout->addWidget(editField->widget());
    }
    QWidget* labels = new QWidget();
    labels->setLayout(retval->m_LabelLayout);
    QWidget* edits = new QWidget();
    edits->setLayout(retval->m_EditLayout);
    retval->addWidget(labels);
    retval->addWidget(edits);
    return retval;
}
}
```

❶ This is a factory method that returns polymorphic concrete instances.

The specific `InputField` types that get created depend on the type of the `Question` passed in, and that is determined in `FormFactory`, shown in Example 17.7.

EXAMPLE 17.7 `src/libs/forms/formfactory.cpp`

```
[ . . . . ]

InputField* FormFactory::createEditor(Question* q) {
    QVariant::Type type = q->type();
    InputField* retval = 0;
    switch(type) {
    case QVariant::StringList:
        retval = new ChoiceInputField(
            q->objectName(), q->choices());
    }
```

continued

```

        break;
    case QVariant::String:
        retval = new StringInputField(q->objectName());
        break;
    case QVariant::Int:
        retval = new IntInputField(q->objectName());
        break;
    case Variant::Dir:
        retval = new DirInputField(q->objectName());
        break;
    default:
        retval=new StringInputField(q->objectName(), 0);
        qDebug() << QString("Unknown property type %1").arg(type);
    }
    if (q->flags() != Qt::ItemIsEditable) {
        retval->setReadOnly(true);
    }
    return retval;
}

```

In Example 17.7, notice the `switch` statement, which is normally to be avoided in object-oriented code. We have it here to map polymorphically from the `QVariant::Type` (an enumerated value) to an `InputField` class. This makes it possible for us to use the Strategy pattern on `InputField` (which provides input and output in various ways, on various types).

By default, `createEditor()` returns a `StringInputField`, shown in Example 17.8. It has a simple `QLineEdit` as its input widget.

EXAMPLE 17.8 `src/libs/forms/inputfields.h`

```

[ . . . . ]

class StringInputField : public InputField {
    Q_OBJECT
public:
    StringInputField(QString name, QWidget* parent = 0);
    QVariant value() const ;
    QWidget* widget() const ;
public slots:
    void setReadOnly(bool v);
    void setView(QVariant qv);
    void clearView();
protected:
    QLineEdit *qle;
};

```

17.2.3 Unforseen Types

It is possible there will be other “types” of data that correspond to different kinds of input widgets, but are not among those defined in `QVariant`. In Example 17.9, we introduce user-defined enumerated values above `QVariant` (127) that will not share a value with those already predefined in `QVariant::Type`.

EXAMPLE 17.9 `src/libs/dataobjects/variant.h`

```
#ifndef VARIANT_H
#define VARIANT_H
#include <QVariant>

namespace Variant {
    const QVariant::Type File = static_cast<QVariant::Type>(128);
    const QVariant::Type Dir = static_cast<QVariant::Type>(129);
}

#endif
```

A directory can be encoded and decoded as a `QString` quite naturally, but a `Question` with a `Variant::Directory` as its type gives a hint to the `FormFactory` that the input widget it creates should be a `QFileDialog` that is already in “directory-chooser” mode.

EXAMPLE 17.10 `src/libs/forms/dirinputfield.h`

```
[ . . . . ]
class DirInputField : public StringField {
    Q_OBJECT
public:
    DirInputField(QString name);
    QWidget* widget() const;
    void clearView();
    static void setFileDialog(QFileDialog* fd) {
        sFileDialog = fd;
    }
public slots:
    void browse();
private:
    QHBoxLayout *m_Layout;
    QPushButton *m_Button;
    QWidget *m_Widget;
    static QFileDialog* sFileDialog;
};
[ . . . . ]
```

The `DirInputField`, defined in Example 17.10, extends the `StringInputField` and still has the `QLineEdit` for accepting a string from the user. In addition, there is a `Browse` button, which when clicked will pop up a `QFileDialog` pre-set to accept only a directory as a valid selection.

17.2.4 Controlling Actions

In this section, we discuss issues of synchronizing data between the model and the view. Since these methods depend on both model and view, we are going to isolate them from both, in their own controller classes. In Example 17.11, we derived two custom `QAction` classes, each responsible for synchronizing in one direction.

EXAMPLE 17.11 `src/libs/forms/formactions.h`

```
[ . . . . ]
class OkAction : public QAction {
    Q_OBJECT
public:
    OkAction(FormModel* model, FormView* view);
public slots:
    void ok();
private:
    FormModel *m_Model;
    FormView *m_View;
};

class CancelAction : public QAction {
    Q_OBJECT
public slots:
    void cancel();
[ . . . . ]
```

`OkAction` (or `apply`) should send the data from the view to the model. `CancelAction`, in the case where the dialog is not to be closed afterwards, should do the opposite (send the data from the model back to the view, to restore old or set default values). Their definitions are in Example 17.12.

EXAMPLE 17.12 `src/libs/forms/formactions.cpp`

```
#include <QDebug>
#include "formactions.h"
#include "formmodel.h"
#include "formview.h"
#include "inputfield.h"
#include "question.h"
```



```

OkAction::OkAction(FormModel* model, FormView* view) :
    QAction( tr("&Ok"), view), m_Model(model), m_View(view) {
    connect (this, SIGNAL(triggered()), this, SLOT(ok()));
}

void OkAction::ok() {
    qDebug() << "OK()" << endl;
    QList<QVariant> values;
    InputList fields = m_View->fields();
    foreach (InputField* field, fields) {
        QVariant v = field->value();
        qDebug() << "submitting value: " << v.toString();
        values += v;
    }
    m_Model->setValues(values);
    qDebug() << m_Model->toString();
}

CancelAction::CancelAction(FormModel* model, FormView* view) :
    QAction( tr("&Cancel"), view), m_Model(model), m_View(view) {
    connect (this, SIGNAL(triggered()), this, SLOT(cancel()));
}

void CancelAction::cancel() {
    qDebug() << "Cancel() " << endl;
    QList<Question*> qlist = m_Model->questions();
    InputList fields = m_View->fields();
    for (int i=qlist.size()-1; i>=0; --i) {
        Question* q = qlist.at(i);
        InputField* f = fields.at(i);
        qDebug() << QString(" name: %1 val: %2")
            .arg(q->objectName())
            .arg(q->value().toString());
        f->setView(q->value());
    }
}

```

These actions are in fact **delegates**, and perform a similar function to Qt's `QItemDelegate`.

17.2.5 DataObject Form Model

In Example 17.1, we extended `FormModel`, and in the constructor we created and added `Question` objects to compose a custom form. The `FormModel` itself can be used in other ways, including those listed below.

1. Creating a `FormModel` from a `DataObject` (one `Question` per property)
2. Connecting fields of a `FormModel` to `QSettings` values, to give persistence

3. Importing and exporting in XML³
4. Importing and exporting in HTML⁴
5. Can you think of others?

We wish to create a `FormModel` from a `DataObject`, so this means that another function goes into the `ModelFactory`. We extended `Question`, the basic `FormModel` building block, so that it would get/set values from/to a `DataObject` property instead of its own `m_Value`.

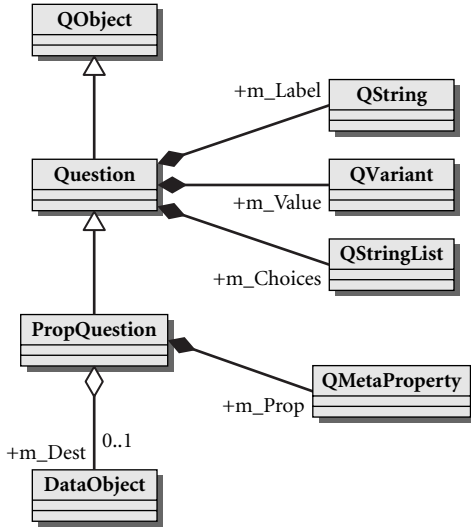


FIGURE 17.6 Rephrasing the question

`PropQuestion`, shown in Figure 17.6, serves as a *proxy* or *delegate* between a `DataObject` property and an `InputField` widget. We see this in Example 17.13: Most of the methods simply pass on the request to the underlying `DataObject`, `mDest`.

EXAMPLE 17.13 `src/libs/forms/propquestion.cpp`

```

#include "propquestion.h"
#include <QMetaProperty>
#include <QVariant>

PropQuestion::PropQuestion(QString name, DataObject* dest):
    m_Dest(dest) {
    setObjectName(name);
  
```

³ This is how Designer recreates its GUIs.

⁴ Basically, this is similar to importing and exporting in XML, except that the tags we use, `<form>` and `<input>`, are specified by the W3C [w3c]. By using this format, we can create XHTML forms and load them in as `FormModels`.

```

    m_Prop = m_Dest->metaProperty(name);
    setLabel(name);
    setType(m_Prop.type());
}

Qt::ItemFlags PropQuestion::flags() const {
    if (m_Prop.isWritable()) return Qt::ItemIsEditable;
    else return Qt::ItemIsSelectable;
}

QVariant PropQuestion::value() const {
    return m_Dest->property(objectName());
}

bool PropQuestion::setValue(QVariant newValue) {
    return m_Dest->setProperty(objectName(), newValue);
}

```

Example 17.14 uses the `DataObject` model applied to the `FileTagger` class to auto-generate a form, which looks like Figure 17.7.

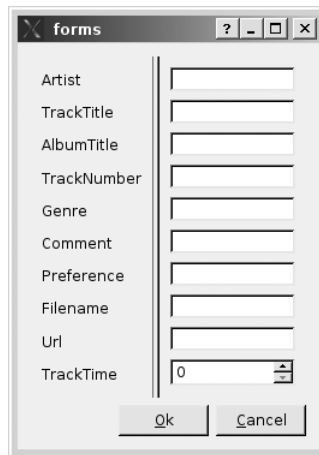


FIGURE 17.7 Auto-generated FileTagger form

EXAMPLE 17.14 `src/libs/forms/testform2.cpp`

```

#include <QMainWindow>
#include <QApplication>
#include "formfactory.h"
#include "formdialog.h"
#include "formmodel.h"
#include "filetagger.h"

int main(int argc, char** argv) {
    QApplication a(argc, argv);
    QMainWindow mw;

```

continued

```

FileTagger ft;
FormModel* mod = FormFactory::newForm(&ft);
FormDialog dialog(mod);
mw.setCentralWidget(&dialog);
mw.setVisible(true);
return a.exec();
}

```

The `newForm` Factory method defined in Example 17.15 simply returns a `PropQuestion` instead of a `Question` when it is creating the `FormModel` from a `DataObject`.

EXAMPLE 17.15 `src/libs/forms/formfactory.cpp`

```

[ . . . . ]

FormModel* FormFactory::newForm(DataObject* dobj) {
    QStringList props = dobj->propertyNames();
    FormModel *mod = new FormModel(dobj->className());
    foreach (QString prop, props) {
        if (prop == "objectName")
            continue;
        PropQuestion *pq = new PropQuestion(prop, dobj);
        *mod << pq;
    }
    return mod;
}

```

EXERCISES: DATAOBJECT FORM MODEL

1. Example 17.16 is an XHTML [w3c] fragment that contains three different kinds of input widgets and roughly represents the form we've seen earlier.

EXAMPLE 17.16 `src/modelview/html/bridgekeeper.html`

```

<form>
<title> I am the keeper of the bridge of death </title>
<p> Answer these questions three and you can proceed over the
bridge. </p>
<label for="name">What is your name? </label>
<input name="name" type="text">

<label for="quest">What is your quest? </label>
<input name="quest" type="text" />

<label for="color">What is your favorite color? </label>
<select name="color">
    <option value="blue">blue</option>
    <option value="green">green</option>

```

```
<option value="orange">orange</option>
<option value="burgundy">burgundy</option>
<option value="crimson">crimson</option>
</select>

<input type="submit" name="ok" value="ok" />
<input type="submit" name="cancel" value="cancel" />
</form>
```

It is possible to preview it in a browser by opening it as a file. It doesn't look fancy without any CSS styling, but you can use it as a sanity check for your files.

Write a `FormReader` class that can read an XML file of the above format. (Do not worry about handling XHTML elements or formats that are not shown in Example 17.16.)

2. Write a `FormWriter` class that can write a `FormModel` to an XML file in the same format.
3. Write a Mad Libs game that asks the user for a bunch of nouns, verbs, adjectives, and adverbs such that when the form is submitted, it sticks the strings into a paragraph and shows the result to the user. The passage of text should be at least two paragraphs long, and contain at least ten blanks to be filled in.

17.3 Qt 4 Models and Views

Qt 4 offers general-purpose view classes for the most common types of views: lists, trees, and tables. This includes abstract and concrete data models that can be extended and customized to hold different kinds of data.

Figure 17.8 shows the four main types of classes in the Qt 4 model-view framework. Each class has a specific role.

1. Item models are objects for representing a data model for multiple items. Item models store the actual data that is to be viewed/manipulated.
2. Views are objects for acquiring, changing, and displaying the data. Each view holds a pointer to a model. View classes make frequent calls to item model methods to get and set data.
3. Selection models are objects that describe which items in the model are selected in the view. Each view has a selection model.
4. `QModelIndex` acts like a cursor, or a smart pointer, providing a uniform way to iterate through list, tree, or table items inside the model.

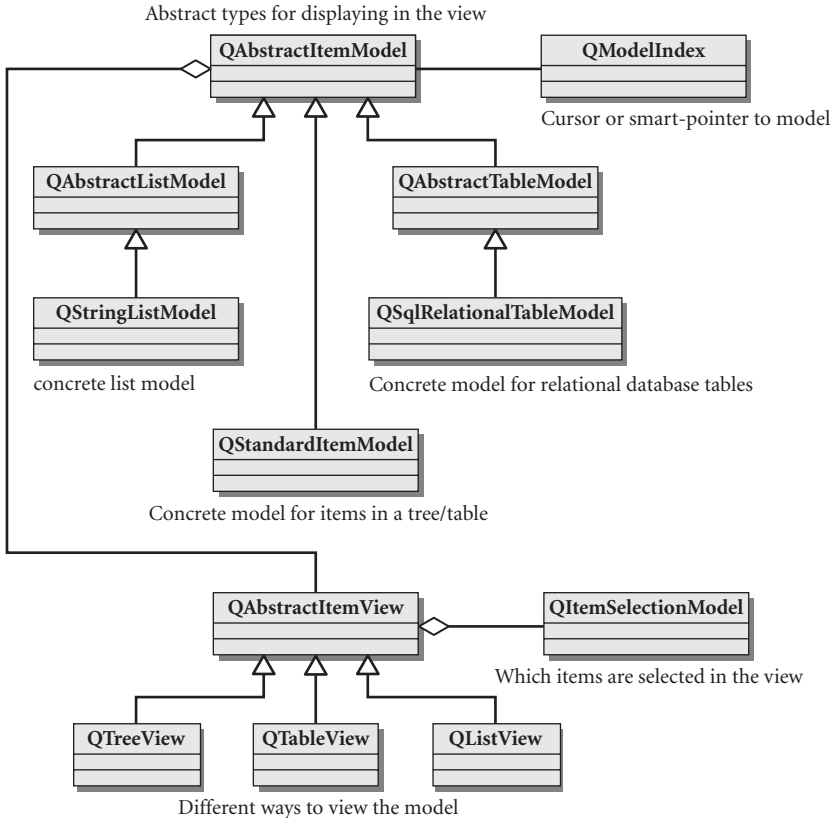


FIGURE 17.8 Qt 4 Model/View classes

The code fragment below shows how to create and connect a view to a model.

```

class MyTableModel : public QDefaultTableModel    {...};
// ...
myModel = new MyTableModel();
myView = new QTableView();
myView->setModel(myModel);

```

After `setModel()` is called, the view should automatically update itself whenever the model changes (assuming the model is written properly).

As we write the implementation of `MyTableModel`, we are once again implementing a *passive interface*, and there is an inversion of control. All the methods we override in `MyTableModel` are called from `QTableView` or `QModelIndex`.



ABSTRACT OR DEFAULT? When extending `QAbstractxxxModel`, derived classes need to override *all* of the pure virtual methods and define a full implementation for the model. In contrast, by extending one of the `QDefaultxxxModel` classes, a derived class inherits a default implementation that does not require overrides of all methods. Each method has an empty stub in the default base class.

Views

A View class encapsulates the components of a graphical user interface that accesses the data in a model. Views come in a variety of different sizes and shapes. In general, they are usually

- Lists in various arrangements
- Tables, perhaps with interactive elements
- Trees representing objects in a parent-child hierarchy
- Graphs and charts

Model Index

The `QModelIndex` class provides a generic access system that works for all classes derived from `QAbstractItemModel`. This system treats model data as if it were arranged in a rectangular array with row and column indices, regardless of what underlying data structure actually holds the data.

`QModelIndex` objects, created by the model, can be used by model, view, or delegate code to locate particular items in the data model. `QModelIndex` objects have short life spans and can become invalid shortly after being created, so they should be used immediately and then discarded.

`QModelIndex::isValid()` should be called before using a `QModelIndex` object that has existed for more than a few operations. `QPersistentModelIndex` objects have longer life spans but still should be checked with `isValid()` before being used.

17.4 Table Models

It would be useful to have an editable `QTableView` for viewing and editing a collection of `DataObjects`. For this, we extend and customize the `QAbstractTableModel`. See Example 17.17.

EXAMPLE 17.17 src/libs/dataobjects/dataobjecttablemodel.h

```

[ . . . . ]

class DataObjectTableModel : public QAbstractTableModel {
    Q_OBJECT
public:
    DataObjectTableModel(DataObject* headerModel = 0);
    virtual DataObject* record(int rowNum) const ;
    virtual bool insertRecord(DataObject* newRecord,
                              int position = -1,
                              const QModelIndex& = QModelIndex());
    QStringList toStringList() const;
    QString toString() const;
    virtual int fieldIndex(QString fieldName) const;
    virtual ~DataObjectTableModel();
[ . . . . ]

public slots:
    void reset();
    void checkDirty();
protected slots:
    void changeProperty(const QString&, const QVariant&);

protected:
    QList<DataObject*> m_Data;
    QStringList m_Headers;
    DataObject* m_Original;
    QTimer m_Timer;
    bool m_Dirty;
    void extractHeaders(DataObject* hmodel);
public:
    DataObjectTableModel& operator<<(DataObject* newObj) {
        insertRecord(newObj);
        return *this;
    }
};

```

	objectName	Id	DateEstablished	Type
Row 0	luke skywal...	14123		Corporate
Row 1	Ben Kenobi	1231		Corporate
Row 2	Princess Leia			Individual

The name, `DataObjectTableModel`, is quite self-descriptive: a table of `DataObjects`. Using this class to group `DataObjects` makes creating editable views reasonably simple. Example 17.18 shows the client code that produced the screenshot above.

EXAMPLE 17.18 src/modelview/tablemodel/tablemodel.cpp

```

#include "dataobjecttablemodel.h"
#include "customerfactory.h"
#include "country.h"

DataObjectTableModel* model() {
    CustomerFactory* fac = CustomerFactory::instance();
    Customer* cust1 = fac->newCustomer("luke skywalker", Country::USA);
    DataObjectTableModel* retval = new
    DataObjectTableModel(cust1); ❶
    cust1->setId("14123");
    *retval << cust1; ❷
    *retval << fac->newCustomer("Ben Kenobi", Country::Canada);
    *retval << fac->newCustomer("Princess Leia", Country::USA);
    return retval;
}

#include <QTableView>
#include <QApplication>
#include <QMainWindow>
#include <QDebug>

int main(int argc, char** argv) {
    QApplication app(argc, argv);
    DataObjectTableModel *mod = model();
    QMainWindow mainwin;
    QTableView view ;
    view.setModel(mod);
    mainwin.setCentralWidget(&view);
    mainwin.setVisible(true);
    int retval = app.exec();
    qDebug() << "Application Exited. " << endl;
    qDebug() << mod->toString() << endl;
    delete mod;
    return retval;
}

```

- ❶ header model
- ❷ Insert row into table.

In the public interface, we want convenient functions for operating on rows as `DataObject` records. See Example 17.19.

EXAMPLE 17.19 src/libs/dataobjects/dataobjecttablemodel.cpp

```

[ . . . . ]

bool DataObjectTableModel::
insertRecord(DataObject* newRow, int position,
            const QModelIndex &parent) {

```

continued

```

    if (position==-1)
        position=rowCount()-1;
    connect (newRow, SIGNAL(propertyChanged(const QString&,
                                           const QVariant&)),
            this, SLOT(changeProperty(const QString&,
                                      const QVariant&)));
    beginInsertRows(parent, position, position);
    m_Data.insert(position, newRow);
    endInsertRows();
    return true;
}

DataObject* DataObjectTableModel::
record(int rowNum) const {
    return m_Data.at(rowNum);
}

```

But How Does It Work?

`QAbstractTableModel` has a series of pure virtual functions, declared in Example 17.20, which *must* be overridden, because they are invoked by `QTableView` to get and set data.

EXAMPLE 17.20 `src/libs/dataobjects/dataobjecttablemodel.h`

```

[ . . . . ]

/* Methods which are required to be overridden
   because of QAbstractTableModel */
int rowCount(const QModelIndex& parent = QModelIndex()) const;
int columnCount(const QModelIndex& parent = QModelIndex())
    const;
QVariant data(const QModelIndex& index, int role) const;
QVariant headerData(int section, Qt::Orientation orientation,
                    int role = DisplayRole) const;
ItemFlags flags(const QModelIndex &index) const;
bool setData(const QModelIndex &index, const QVariant &value,
             int role = EditRole);
bool insertRows(int position, int rows,
                const QModelIndex &index = QModelIndex());
bool removeRows(int position, int rows,
                const QModelIndex &index = QModelIndex());

```

Example 17.21 shows the methods used to get data in and out of the model.

EXAMPLE 17.21 src/libs/dataobjects/dataobjecttablemodel.cpp

```
[ . . . . ]

QVariant DataObjectTableModel::
data(const QModelIndex &index, int role) const {
    if (!index.isValid())
        return QVariant();
    if (role == DisplayRole) {
        int row(index.row()), col(index.column());
        DataObject* listItem(m_Data.at(row));
        return listItem->property(m_Headers.at(col));
    } else
        return QVariant();
}

bool DataObjectTableModel::
setData(const QModelIndex &index, const QVariant &value,
        int role) {
    bool changed=false;
    if (index.isValid() && role == EditRole) {
        int row(index.row()), col(index.column());
        DataObject* listItem(m_Data.at(row));
        changed = listItem->setProperty(m_Headers.at(col), value);
        if(changed)
            emit dataChanged(index, index);
    }
    return changed;
}

```

This is a **mapping layer** from objects to tables. Since the tables need to show header data, the table model has one `DataObject`, designated the *header model*, which it uses to obtain headers. Example 17.22 defines `headerData`, the method that table models call, which we can override to provide the proper header names.

EXAMPLE 17.22 src/libs/dataobjects/dataobjecttablemodel.cpp

```
[ . . . . ]

QVariant DataObjectTableModel::
headerData(int section, Qt::Orientation orientation,
        int role) const {
    if (role != DisplayRole)
        return QVariant();
    if(orientation == Qt::Vertical)
        return QVariant(section);
    if (m_Headers.size() ==0)
        return QVariant();
    return m_Headers.at(section);
}

```

continued

```

int DataObjectTableModel::rowCount(const QModelIndex&) const {
    return m_Data.count();
}

int DataObjectTableModel::
columnCount(const QModelIndex &parent) const {
    Q_UNUSED(parent);
    return m_Headers.size();
}

```

The other methods in the abstract interface, shown in Example 17.23, tell views which items are editable (`flags()`), or allow client code to insert and remove rows.

EXAMPLE 17.23 `src/libs/dataobjects/dataobjecttablemodel.cpp`

```

[ . . . . ]

ItemFlags DataObjectTableModel::
flags(const QModelIndex &index) const {
    if (!index.isValid())
        return ItemIsEnabled;
    // TODO - check the metaProperty to see if it is read/write
    return QAbstractItemModel::flags(index) | ItemIsEditable;
}

void DataObjectTableModel::
reset() {
    QModelIndex br = index(rowCount()-1, columnCount()-1);
    QModelIndex tl = index(0,0);
    emit dataChanged(tl, br);
    m_Dirty = false;
}

bool DataObjectTableModel::
insertRows(int position, int rows, const QModelIndex &parent) {
    beginInsertRows(parent, position, position+rows-1);
    for (int row = 0; row < rows; ++row) {
        DataObject* dobj = m_Original->clone();
        m_Data.insert(position, dobj);
    }
    endInsertRows();
    return true;
}

bool DataObjectTableModel::
removeRows(int position, int rows, const QModelIndex& parent) {
    for (int row = 0; row < rows; ++row) {
        delete m_Data.at(position);
        m_Data.removeAt(position);
    }
    QModelIndex topLeft(index(position, 0, parent));

```

```

    QModelIndex bottomRight(index(position + 1, columnCount(), parent));
    emit dataChanged(topLeft, bottomRight);
    return true;
}

```

17.5 Tree Models

To represent data for a hierarchy of widgets (parents and children), Qt offers two choices of models:

1. `QAbstractItemModel` is a general-purpose, but very complex class, that can be used with `QTreeView` as well as `QListView` and `QTableView`.
2. `QTreeWidgetItem` is a simpler model, specifically for use with `QTreeWidgetItem`.

The `QTreeWidgetItem` class is a tree node that can be instantiated or extended. The widget items need to be connected together in a tree-like fashion, similar to `QObject` children (Section 9.2) or `QDomNodes` (Section 14.3). In fact, `QTreeWidgetItem` is another implementation of the Composite pattern.

EXAMPLE 17.24 `src/widgets/trees/treedemo.cpp`

```

#include <QTreeWidgetItem>
#include <QTreeWidgetItemItem>
#include <QApplication>

QTreeWidgetItem *item(QString name, QTreeWidgetItem* parent=0) {
    QTreeWidgetItem *retval = new QTreeWidgetItem(parent);
    retval->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEditable
        | Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled |
        Qt::ItemIsEnabled);
    retval->setText(0, name);
    return retval;
}

int main(int argc, char** argv) {
    QApplication app(argc, argv);
    QTreeWidgetItem *root = item("root");
    QTreeWidgetItem *colors = item("colors", root);
    item("blue", colors);
    item("red", colors);
    item("orange", colors);

    QTreeWidgetItem *sports = item("sports", root);
    item("baseball", sports);
    item("hockey", sports);
    item("curling", sports);

    QTreeWidgetItem *food = item("food", root);
    item("rutabega", food);
}

```

continued

```

    item("macademia nuts", food);
    item("bok-choy", food);

    QTreeWidget *tree = new QTreeWidget();
    tree->addTopLevelItem(root);
    tree->setColumnCount(1);
    tree->setVisible(true);
    return app.exec();
}

```

Example 17.24 is a simple main program that creates a tree model and a view for it. When you run it, you should see a tree widget that looks like Figure 17.9.

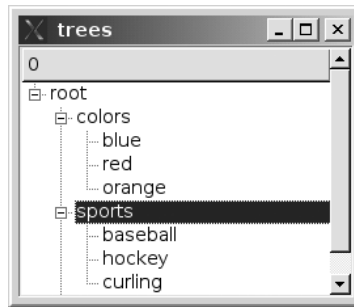
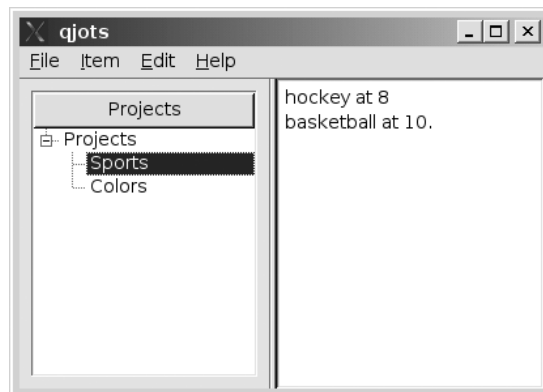


FIGURE 17.9 Tree Widget

Because the items are editable, you can edit them with the F2 key. You might notice that other familiar keyboard shortcuts (cut, copy, paste, ctrl+cursor keys, etc.) from your window environment will also work inside these list widgets and text areas.

17.5.1 Extended Tree Widget Items

This section discusses the design behind the `qjots` application. `qjots` has a `TreeView` on the left, and it allows you to organize your notes and Web bookmarks in a tree-like fashion. The figure that follows demonstrates this.



The main idea behind this application is that whenever an item is selected in the tree on the left, the appropriate view should be shown on the right. This is actually a typical feature for most GUI applications.

The `QTreeWidgetItem` is a natural base class to extend for abstract data types. However, it is not a `QObject`, so it does not support signals or slots (which we need on this model). Therefore, we will use multiple inheritance (Section 23.3) to bring both classes together. Notice in Example 17.25 that this class is abstract, because of a pure `virtual` function. We plan to extend this class further for the specific types of data to display.

EXAMPLE 17.25 `src/modelview/qjots/item.h`

```
[ . . . . ]
class Item : public DataObject, public QTreeWidgetItem {
    Q_OBJECT
    Q_PROPERTY( QString Name READ name WRITE setName );
public:
    Item();
    virtual ~Item() {}
    QTreeWidgetItem* parent() const { ❶
        return QTreeWidgetItem::parent();
    }
    virtual void setName(QString name) ;
    virtual QDomElement element( QDomDocument doc)=0;
    virtual QWidget* detailView() =0;
    virtual QString name() const ;
};
[ . . . . ]
```

❶ Required to eliminate the conflict between the two base class versions.

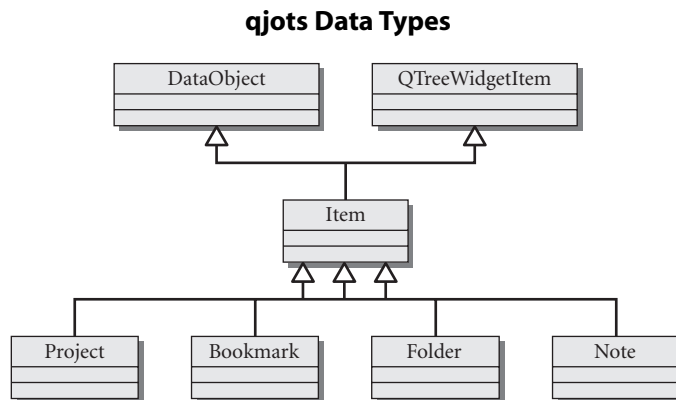
Notice that `parent()` must be defined in `Item` to eliminate the conflict between the two base class definitions. What is nice about extending the `Item` class is that you can initialize each instance with default flags, like we see in Example 17.26.

EXAMPLE 17.26 `src/modelview/qjots/item.cpp`

```
[ . . . . ]
Item::Item() {
    setFlags(Qt::ItemIsSelectable | Qt::ItemIsEditable
            | Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled
            | Qt::ItemIsEnabled );
}
[ . . . . ]
```

EXERCISES: TREE MODELS

Your assignment is to write a `qjots` application that permits the user to create, select, view, and edit at least three different “kinds” of items in a `TreeWidget`. If you want, you can use bookmarks, notes, and folders, as shown in the diagram below. Alternately, you can make up your own data model or adapt one from a previous assignment.



1. Use a `QSplitter` to separate the left and the right sides of your `QMainWindow`.
2. Write serializers so that the entire tree can be loaded and saved to disk.
3. Allow the user to select items in a tree on the left, so that they are visible/editable in a “detailed view” on the right side.
4. Allow the user to add and delete items from the tree.

REVIEW QUESTIONS

1. What is controller code? Which Qt classes are controller classes?
2. What pattern(s) is/are used in the design of `InputField`?
3. Because there is a 1:1 correspondence between `InputField` and `Question`, we could easily combine the two classes into one. What would you call that class? Explain the advantages or disadvantages of this design.
4. How do you determine what item(s) is/are selected in a `QListView`?
5. If we wanted to iterate through items in an `QAbstractItemModel`, what would be a good class to use?
6. There are two distinct model-view class pairs for storing and displaying tree-like data. What are they called? Why would you use one versus the other?


18

CHAPTER 18

Qt SQL Classes

This chapter gives a general introduction to the capabilities of Qt's SQL classes, using MySQL as an example back end.

18.1 Introduction to MySQL	424
18.2 Queries and Result Sets	427
18.3 Database Models	429



Qt 4 provides a platform-neutral database interface similar to JDBC but without the annoying mandatory exception-handling code. You can use Qt to connect to a variety of different SQL databases, including Oracle, PostgreSQL, and SybaseSQL. In the examples that follow, we use MySQL¹ because it

1. Is open source
2. Is available on all platforms
3. Comes pre-installed on most Linux distributions
4. Has excellent documentation
5. Is very widely used

18.1 Introduction to MySQL

After you have installed MySQL on your system, you can create a database from its shell by entering `mysql` as the “root” or admin user, as shown here.²

```
/home/files> mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

Next, cut and paste the following lines from Example 18.1 into the MySQL shell.

EXAMPLE 18.1 `src/sql/dbcreate.sql`

```
create database mp3db;
use mp3db;
grant all on mp3db.* to 'mp3user'@'localhost' identified by 'mp3dbpw';
grant all on mp3db.* to mp3user identified by 'mp3dbpw';
```

¹ <http://dev.mysql.com/doc/>

² If you did not install MySQL yourself, then you may need to ask a system admin to create a MySQL account for you.

The last two lines grant user permissions for `mp3user`, which is the `userid` we will use for all subsequent operations on the data.

Connecting to MySQL from Qt

Connecting initially to the database requires four pieces of information: user, password, database, and host, as shown in Example 18.2.

EXAMPLE 18.2 `src/sql/mp3db.cpp`

```
[ . . . . ]

bool Mp3db::connect ()
{
    QSqlDatabase db;
    db = QSqlDatabase::addDatabase("QMYSQL");
    db.setDatabaseName("mp3user");
    db.setUserName("mp3db");
    db.setPassword("mp3dbpw");
    if (!db.open()) {
        qCritical("Cannot open database: %s (%s)",
                db.lastError().text().toLatin1().data(),
                qt_error_string().toLocal8Bit().data());
        return false;
    }
    return true;
}
```

After the database connection has been opened, we use a very powerful class called `QSqlQuery`, which has a member function `exec()` that enables us to submit standard SQL statements to the database.

Defining a Table

Each database has a collection of **tables**. A table is very much like an array of `struct`, where each column corresponds to a data member. To define a table, we must describe each of the columns, which can also be thought of as fields, properties, or data members.

EXAMPLE 18.3 `src/sql/filetagger.sql`

```
CREATE TABLE FileTagger (
  Artist      varchar(100),
  TrackTitle  varchar(100),
  AlbumTitle  varchar(100),
  TrackNumber varchar(10),
  Genre       varchar(20),
  Comment     varchar(200),
  Preference  varchar(20),
  Filename    varchar(200),
  PRIMARY KEY(Filename),
  INDEX(Preference),
  INDEX(Genre)
);
```

Example 18.3 defines a single table in SQL called `FileTagger`. This table has a structure that includes columns for each of the properties of the `FileTagger` class. There are three ways you can create this table.

1. Cut and paste the contents of Example 18.3 into the `mysql` shell.
2. Source the file from the `mysql` shell.


```
> mysql mp3db -u mp3user -p
Enter password: XXXXX
mysql> source filetagger.sql
```
3. Pass it as a string to `QSqlQuery::exec()`.



If you don't want to keep reentering the db/user/pw each time you run `mysql`'s command line shell, you can set default values in `~/ .my.cnf` (*nix) or `c:\mysql\my.cnf` (Win32).

```
[mysql]
user=mp3user
password=mp3dbpw
database=mp3db
```

Inserting Rows

We wish to extract data from the ID3 tags of MP3 files and import them into the `FileTagger` table. The first step is to prepare an SQL statement for inserting the data, as shown in Example 18.4. **Prepared statements** are useful when we must

execute the same SQL statement repeatedly—the server only needs to parse the string once.

EXAMPLE 18.4 `src/sql/mp3db.cpp`

```
[ . . . . ]
Mp3db::Mp3db() {
    connect();
    m_insertQuery.prepare("INSERT INTO FileTagger ("
                          "Artist, TrackTitle, AlbumTitle,
                          TrackNumber, Genre, "
                          "Comment, Preference, Filename) VALUES
                          (?, ?, ?, ?, ?, ?, ?, ?)");
}
```

We left the ? character in the parts of the SQL prepared statement where we later wish to bind values. When we have a `FileTagger` object with the desired data to import, we call `addFile()`, shown in Example 18.5. This method binds the values into the prepared statement.

EXAMPLE 18.5 `src/sql/mp3db.cpp`

```
[ . . . . ]
void Mp3db::addFile(FileTagger* song) {
    m_insertQuery.addBindValue(song->getArtist());
    m_insertQuery.addBindValue(song->getTrackTitle());
    m_insertQuery.addBindValue(song->getAlbumTitle());
    m_insertQuery.addBindValue(song->getTrackNumber());
    m_insertQuery.addBindValue(song->getGenre());
    m_insertQuery.addBindValue(song->getComment());
    m_insertQuery.addBindValue(song->getPreference());
    m_insertQuery.addBindValue(song->getFilename());
    m_insertQuery.exec();
}
```

18.2 Queries and Result Sets

In Example 18.6, we define a function that queries an existing `amaroK`³ MySQL database. `amaroK` is a jukebox program for KDE that can use a variety of different back ends for storing its Mp3 metadata. Instead of storing all tag data for a song

³ <http://amarok.kde.org/>

in a single row, like we did in Example 18.5, amaroK spreads the tag data across many tables with relatively fewer rows. `import ()` will query the “tags” table and, for each URL, insert a row of preference data into the “statistics” table.

EXAMPLE 18.6 `src/mmjbamarok/tool.cpp`

```
[ . . . . ]
void import() {
    using namespace qstd;
    RatingMapper mapper;
    FileTagger ft;
    QSqlDatabase db = connect();

    QSqlQuery insert;
    int entries=0;
    insert.prepare("INSERT INTO statistics (rating, url)"
        " VALUES (?, ?) "
        " ON DUPLICATE KEY UPDATE rating=? "); ❶
    QSqlQuery query;
    query.exec("select url from tags");        ❷
    while (query.next()) {                    ❸
        QString filename = query.value(0).toString();
        ft.setFilename(filename);
        QString preference = ft.getPreference();
        int rating = mapper.toRating(preference);
        if (rating == 0) continue;
        insert.addBindValue(rating);          ❹
        insert.addBindValue(filename);
        insert.addBindValue(rating);
        if (insert.exec()) {
            cerr << rating << " : " << filename << endl;
            entries++;
        }
        else {
            cerr << "Error inserting " << filename << endl;
        }
    }
    cerr << "Entries imported:" << entries << endl;
}
}
```

- ❶ Prepare an SQL statement that inserts a new (rating,url) record or updates the current record (new rating) if the record is already there. Each ? is a positional parameter to which we later `addBindValue()`.
- ❷ Find all urls in the “tags” table. There is one for each song.
- ❸ Iterate through result set.
- ❹ first positional parameter

In this example, a JDBC programmer might observe that `QSqlQuery` serves the purpose of at least two JDBC classes. It is being used in two ways.

1. As a `PreparedStatement`—something to store the query, add bind values to, etc.
2. As a cursor into the `ResultSet`—something to iterate through the query results

The function iterates through all songs that are in amaroK’s library. For each song, it extracts the preference string from a `MusicMatch ID3v2` tag and, with the help of `RatingMapper`, translates the preference string into an integer number of “stars.” Finally, the number of stars is inserted into amaroK’s statistics table, under the “rating” column.

18.3 Database Models

Qt 4 provides database model classes that extend the `QAbstractTableModel` but operate on SQL data instead of objects in memory.

Figure 18.1 shows the UML design for a minimal application that displays a view of a table. The `DbViewApp` is responsible for initializing and connecting the objects in our application. We will create a `QTableView` of an SQL table called “FileTagger.” Example 18.7 establishes the database connection.

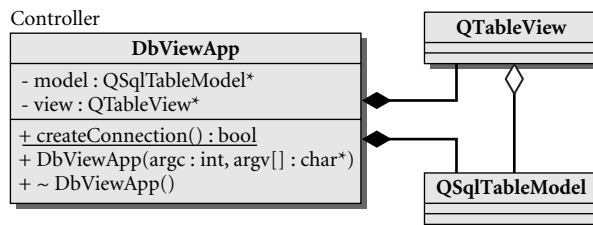


FIGURE 18.1 `DbViewApp`

EXAMPLE 18.7 `src/modelview/database/dbviewapp.cpp`

```
[ . . . . ]

//#include "mp3tablemodel.h"

#include "dbviewapp.h"
#include <QSqlTableModel>
#include <QSqlRelationalDelegate>
#include <QDebug>
#include <QTableView>
#include <QSqlTableModel>
#include <QSqlError>

bool DbViewApp::createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setDatabaseName("mp3db");
    db.setUserName("mp3user");
    db.setPassword("mp3dbpw"); ❶

    if (!db.open()) {
        qCritical("Cannot open database: %s (%s)",
                db.lastError().text().toLatin1().data(),
                qt_error_string().toLocal8Bit().data());
        return false;
    }
    else {
        qDebug() << "Database Opened";
        return true;
    }
}
}
```

❶ It would be better to get this information from an encrypted file, rather than hardcoded in the source.

By setting an `EditStrategy` on the model, it is possible for views of the data to provide editable text fields. `OnManualSubmit` means that the changes that are made in the view are not sent to the server until `submitAll()` is called.

`QTableView` has functions that allow us to customize the selection behavior and mode for the view. The parameters we set, shown in Example 18.8, permit only a single row selection.

EXAMPLE 18.8 `src/modelview/database/dbviewapp.cpp`

```
[ . . . . ]

DbViewApp::DbViewApp(int argc, char* argv[]) :
    QApplication(argc, argv) {

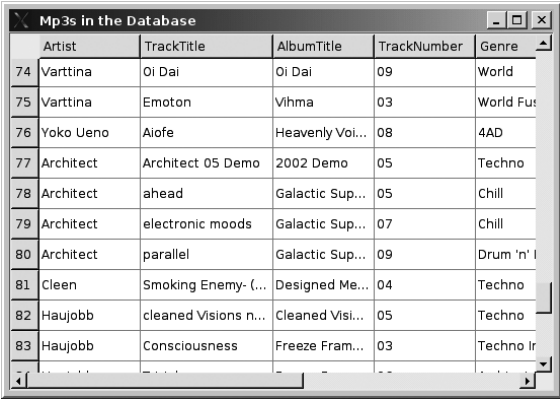
    createConnection(); ❶
    model = new QSqlTableModel(this);
    model->setTable("FileTagger");
    model->setEditStrategy(QSqlTableModel::OnManualSubmit);

    view = new QTableView();
    view->setWindowTitle("Mp3s in the Database");
    view->setModel(model);
    view->setSelectionBehavior(QAbstractItemView::SelectRows);
    view->setSelectionMode(QTableView::SingleSelection);
    model->select();
    view->setVisible(true);
    qDebug() << "DbViewApp()";
}

```

❶ The connection must be created before any models are created.

When you run the application, it will permit viewing and editing of the `FileTagger` SQL table of the `mp3db` database, as shown in the following screenshot.



	Artist	TrackTitle	AlbumTitle	TrackNumber	Genre
74	Varttina	Oi Dai	Oi Dai	09	World
75	Varttina	Emoton	Vihma	03	World Fu...
76	Yoko Ueno	Aiofe	Heavenly Voi...	08	4AD
77	Architect	Architect 05 Demo	2002 Demo	05	Techno
78	Architect	ahead	Galactic Sup...	05	Chill
79	Architect	electronic moods	Galactic Sup...	07	Chill
80	Architect	parallel	Galactic Sup...	09	Drum 'n' I
81	Clean	Smoking Enemy- (...	Designed Me...	04	Techno
82	Haujobb	cleaned Visions n...	Cleaned Visi...	05	Techno
83	Haujobb	Consciousness	Freeze Fram...	03	Techno Ir

Because there is no “submit” button, there is no way to actually change the data, so changes to text fields will not be persistent.

There is no actual SQL code in Example 18.8, because the Qt class provides a higher-level interface. The model class performs database reflection to determine which columns to display.

REVIEW QUESTIONS

1. `QSqlQuery` is a multipurpose class. Explain some of the ways it can be used.
2. What view classes are suitable for displaying a database model?



P A R T I I I

**C++ Language
Reference**



Chapter 19. Types and Expressions	437
Chapter 20. Scope and Storage Class	463
Chapter 21. Statements and Control	
Structures	479
Chapter 22. Memory Access	503
Chapter 23. Inheritance in Detail	523
Chapter 24. Miscellaneous Topics	541


19

CHAPTER 19

Types and Expressions

This chapter seeks to provide a deeper understanding of C++'s strong typing system, and shows how expressions are evaluated and converted.

19.1	Operators	438
19.2	Evaluation of Logical Expressions	443
19.3	Enumerations	443
19.4	Signed and Unsigned	
	Integral Types	445
19.5	Standard Expression Conversions	447
19.6	Explicit Conversions.....	449
19.7	Safer Typecasting Using	
	ANSI C++ Typecasts	450
19.8	Run-Time Type	
	Identification (RTTI)	454
19.9	Member Selection Operators	457



Here we formally define some terms that we have been using. **Operators** are special kinds of functions that perform calculations on operands and return results. **Operands** are the arguments supplied to the operator.

Operators can be thought of as ordinary functions, except that it is possible to call them using operator symbols (e.g., +, -, *, /, etc.) in addition to the longer function-call syntax.

An **expression** consists of a single operand, multiple operands with operators interspersed, or functions with arguments. Each expression has a type and a value. The value is obtained by applying the definitions of the operators and/or functions to the operands (and/or arguments).

19.1 Operators

Operators fall into broad classifications according to their primary use.

Assignment operators	=, +=, *=, ...
Arithmetic operators	+, -, *, /, %
Relational operators	<, <=, >, >=, ==, !=
Logical operators	&&, , !
Bitwise operators	&, , ^, ~, <<, >>
Memory management operators	new, delete, sizeof
Pointer and access operators	*, &, ., ->, []
Scope resolution operators	::
Miscellaneous operators	conditional (? :), comma (,)

Operators have predefined meanings for built-in types, but not all operators are defined for all built-in types.

Operator Characteristics

Operators have the following special characteristics:

- Precedence
- Associativity
- Number of required operands

Table 19.1 lists all the C++ operators and their characteristics, grouped by precedence and purpose, with groups of highest precedence listed first.

- The Operands column contains the number of operands that the operator requires.
- The Description column contains the conventional meaning of the operator for built-in types.
- The Assoc column indicates the associativity that governs how an expression is evaluated if the same operator occurs more than once.
 - L indicates left-to-right associativity. Example:

```
d = a + b + c; // a+b is evaluated first, then (a+b)+c
assignment is evaluated last because of lower precedence.
```
 - R indicates right-to-left associativity:

```
c = b = a; // a is assigned to b, then to c.
```
- The Ovl column indicates whether or not the operator may be overloaded (redefined) for custom types.
- The possible values for that column are:
 - Y: This operator can be overloaded as a global or member function.
 - M: This operator can be overloaded only as a class member function.
 - N: This operator cannot be overloaded.

TABLE 19.1 C++ Operators

Operator	Operands	Description	Example	Assoc	Ovl
::	one	Global Scope Resolution	:: name	R	N
::	two	Class/namespace scope resolution	<i>className</i> :: <i>memberName</i>	L	N
->	two	Member selector via ptr	<i>ptr</i> -> <i>memberName</i>	L	N
.	two	Member selector via obj	<i>obj</i> . <i>memberName</i>	L	N
->	one	Smart ptr	<i>obj</i> -> <i>member</i>	R	M
[]	two	Subscript operator	<i>ptr</i> [<i>expr</i>]	L	M
()	any ^a	Function call	<i>function</i> (<i>argList</i>)	L	N
()	any	Value construction	<i>className</i> (<i>argList</i>)	L	M
++	one	Post increment	<i>varName</i> ++	R	Y
--	one	Post decrement	<i>varName</i> --	R	Y
typeid	one	Type identification	<i>typeid</i> (<i>type</i>) or <i>typeid</i> (<i>expr</i>)	R	N
dynamic_cast	two	runtime checked conv	<i>dynamic_cast</i> < <i>type</i> >(<i>expr</i>)	L	N
static_cast	two	compile time checked conv	<i>static_cast</i> < <i>type</i> >(<i>expr</i>)	L	N
reinterpret_cast	two	unchecked conv	<i>reinterpret_cast</i> < <i>type</i> >(<i>expr</i>)	L	N
const_cast	two	const conv	<i>const_cast</i> < <i>type</i> >(<i>expr</i>)	L	N
sizeof	one	Size in bytes	<i>sizeof expr</i> or <i>sizeof</i> (<i>type</i>)	R	N
++	one	Pre Increment	++ <i>varName</i>	R	Y
--	one	Pre Decrement	-- <i>varName</i>	R	Y
~	one	Bitwise negation	~ <i>expr</i>	R	Y
!	one	Logical negation	! <i>expr</i>	R	Y
+, -	one	Unary plus, unary minus	+ <i>expr</i> or - <i>expr</i>	R	Y

*	one	Pointer dereference	* ptr	R	Y
&	one	Address-of	& lvalue	R	Y
new	one	Allocate	new type or new type(expr-list)	R	Y
new []	two	Allocate array	new type [size]	L	Y
delete	one	Deallocate	delete ptr	R	Y
delete []	one	Deallocate array	delete [] ptr	R	M
()	two	C-style type cast	(type) expr	R	N ^b
->*	two	Member ptr selector via ptr	ptr->*ptrToMember	L	M
.*	two	Member ptr selector via obj	obj.*ptrToMember	L	N
*	two	Multiply	expr1 * expr2	L	Y
/	two	Divide	expr1 / expr2	L	Y
%	two	Remainder	expr1 % expr2	L	Y
+	two	Add	expr1 + expr2	L	Y
-	two	Subtract	expr1 - expr2	L	Y
<<	two	Bitwise left shift	expr << shiftAmt	L	Y
>>	two	Bitwise right shift	expr >> shiftAmt	L	Y
<	two	Less than	expr1 < expr2	L	Y
<=	two	Less or equal	expr1 <= expr2	L	Y
>	two	Greater	expr1 > expr2	L	Y
>=	two	Greater or equal	expr1 >= expr2	L	Y
==	two	Equal ^c	expr1 == expr2	L	Y
!=	two	Not equal	expr1 != expr2	L	Y
&	two	Bitwise AND	expr1 & expr2	L	Y

continued

TABLE 19.1 (Continued)

Operator	Operands	Description	Example	Assoc	Ovl
<code>^</code>	two	Bitwise XOR (exclusive OR)	<code>expr1 ^ e2</code>	L	Y
<code> </code>	two	Bitwise OR (inclusive OR)	<code>expr1 expr2</code>	L	Y
<code>&&</code>	two	Logical AND	<code>expr1 && expr2</code>	L	Y
<code> </code>	two	Logical OR	<code>expr1 expr2</code>	L	Y
<code>=</code>	two	Assign	<code>expr1 = expr2</code>	R	Y
<code>*=</code>	two	Multiply and assign	<code>expr1 *= expr2</code>	R	Y
<code>/=</code>	two	Divide and assign	<code>expr1 /= expr2</code>	R	Y
<code>%=</code>	two	Modulo and assign	<code>expr1 %= expr2</code>	R	Y
<code>+=</code>	two	Add and assign	<code>expr1 += expr2</code>	R	Y
<code>-=</code>	two	Subtract and assign	<code>expr1 -= expr2</code>	R	Y
<code><<=</code>	two	Left shift and assign	<code>expr1 <<= expr2</code>	R	Y
<code>>>=</code>	two	Right shift and assign	<code>expr1 >>= expr2</code>	R	Y
<code>&=</code>	two	And and assign	<code>expr1 &= expr2</code>	R	Y
<code> =</code>	two	Inclusive or and assign	<code>expr1 = expr2</code>	R	Y
<code>*=</code>	two	Exclusive or and assign	<code>expr1 ^= expr2</code>	R	Y
<code>? :</code>	three	Conditional expression	<code>bool ? expr : expr</code>	L	N
<code>throw</code>	one	Throw exception	<code>throw expr</code>	R	N
<code>,</code>	two	Sequential Evaluation (comma)	<code>expr , expr</code>	L	Y

^a The function call operator may be declared to take any number of operands.

^b The type-cast operator may use constructors or conversion operators to convert custom types.

^c Note that for `float` and `double`, this operator should not be used, as it requires an “exact” match, which is architecture-dependent, and not always reliable.

19.2 Evaluation of Logical Expressions

In C and C++, evaluation of a logical expression stops as soon as the logical value of the entire expression is determined. This shortcut mechanism may leave some operands unevaluated. The value of an expression of the form

```
expr1 && expr2 && ... && exprn
```

is `true` if and only if all of the operands are `true`. If one or more of the operands is `false`, the value of the expression is `false`. Evaluation of the expression proceeds sequentially, from left to right, and is *guaranteed to stop* (and return the value `false`) if it encounters an operand that has the value `false`.

Similarly, an expression of the form

```
expr1 || expr2 || ... || exprn
```

is `false` if and only if all of the operands are `false`. Evaluation of the expression proceeds sequentially, from left to right, and is *guaranteed to stop* (and return the value `true`) if it encounters an operand that has the value `true`.

Programmers often exploit this system with statements like:

```
if( x != 0 && y/x < z) {  
    // do something ...  
}  
else {  
    // do something else ...  
}
```

If `x` were equal to 0, evaluating the second expression would produce a run-time error. Fortunately, that cannot happen.

Logical expressions often make use of both `&&` and `||`. It is important to remember that `&&` has higher precedence than `||`. In other words, `x || y && z` means `x || (y && z)`, not `(x || y) && z`.

19.3 Enumerations

In Chapter 2 we discussed at some length how we can add new types to the C++ language by defining classes. Another way to add new types to C++ deserves some more discussion.

The keyword `enum` is used for assigning integral values to C++ identifiers. For example, when designing data structures that perform bitwise operations, it is very convenient to give names to the various bitmasks. The main purpose for an `enum` is to make the code more readable and, hence, easier to maintain. For example,

```
enum {UNKNOWN, JAN, FEB, MAR };
```

defines three constant identifiers, numbered in ascending order, starting at 0. It is equivalent to

```
enum {UNKNOWN=0, JAN=1, FEB=2, MAR=3};
```

The identifiers, JAN, FEB, and MAR are called **enumerators**.

They can be defined and initialized to arbitrary integer values.

```
enum Ages {manny = 10, moe, jack = 23,
           scooter = jack + 10};
```

If the first enumerator, manny, had not been initialized, it would have been given the value 0. Since we initialized manny to 10 and we did not assign a value to moe, the value of moe is 11. The values of enumerators need not be distinct.

When an enum has a **tag name**, then a new type is defined. For example,

```
enum Winter {JAN=1, FEB, MAR, MARCH = MAR};
```

The name Winter is called a tag name. Now, we can declare variables of type Winter.

```
Winter m = JAN;
int i = JAN; // OK - enum can be implicitly converted to int
m = i;      // ERROR! An explicit cast is required.
m = static_cast<Winter>(i); // OK
i = m;      // OK
m = 4;      // ERROR
```

The tag name and the enumerators must be distinct identifiers within their scope.

Enumerations can be implicitly converted to ordinary integer types, but the reverse is not possible without an explicit cast. Example 19.1 demonstrates the use of enum and also shows how the compiler symbols look when they are printed out.

EXAMPLE 19.1 src/enums/enumtst.cpp

```
#include <iostream>
using namespace std;

int main(int, char** ) {
    enum Signal { off, on } sig;           ❶
    sig = on;
    enum Answer { no, yes, maybe = -1 };  ❷
    Answer ans = no;                       ❸
    enum { lazy, hazy, crazy } why;        ❹
    int i, j = on;                          ❺
    sig = off;
    i = ans;
    // ans = sig;                            ❻
    ans = static_cast<Answer>(sig);         ❼
    ans = (sig ? no : yes);
    why = hazy;
    cout << "sig, ans, i, j, why "
         << sig << ans << i << j << why << endl;
```



```
    return 0;
}
```

Output:

```
OOP> gpp enumtest.cc
a, b, i, j, why 01011
OOP>
```

- ❶ a new type, two new enum identifiers, and a variable definition all in one line
- ❷ just the type/enum definitions
- ❸ an instance of an enum
- ❹ a typeless enum variable
- ❺ An enum can always convert to int.
- ❻ Conversions between enum types cannot be done implicitly.
- ❼ Conversion is okay with a cast.

19.4 Signed and Unsigned Integral Types

This section explains the differences between signed and unsigned integral types.

The underlying binary representation of an object x of any integral type looks like this (assuming n -bit storage):

$$d_{n-1}d_{n-2}\dots d_2d_1d_0$$

where each d_i is either 0 or 1. The computation of the decimal equivalent value of x depends on whether x is an unsigned or signed type. If x is unsigned, the decimal equivalent value is

$$d_{n-1} * 2^{n-1} + d_{n-2} * 2^{n-2} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0$$

The largest (positive) value that can be expressed by an unsigned integer is, therefore,

$$2^n - 1 = 1 * 2^{n-1} + 1 * 2^{n-2} + \dots + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

If x is signed, the decimal equivalent value is

$$d_{n-1} * -(2^{n-1}) + d_{n-2} * 2^{n-2} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0$$

The largest (positive) value that can be expressed by a signed integer is

$$2^{n-1} - 1 = 0 * -(2^{n-1}) + 1 * 2^{n-2} + \dots + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

This is called “two’s complement” representation. To determine the representation of the negative of a signed integer,

1. Compute the “one’s complement” of the number (i.e., replace each bit with its complement).
2. Add 1 to the “one’s complement” produced in the first step.

8-Bit Integer Example

Suppose that we have a tiny system that uses only 8 bits to represent a number. On this system, the largest unsigned integer would be

$$11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

But that same number, interpreted as a signed integer, would be

$$11111111 = -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1$$

EXERCISES: SIGNED AND UNSIGNED INTEGRAL TYPES

You be the computer ...

For these exercises, simulate the action of the computer as it executes the given code and specify what you think the output will be. You can always compile and run the code yourself to see if your output is correct. If you disagree with the computer, try to explain why.

```
1. #include <iostream>
using namespace std;
int main() {
    unsigned n1 = 10;
    unsigned n2 = 9;
    char *cp;
    cp = new char[n2 - n1];
    if(cp == 0)
        cout << "That's all!" << endl;
    cout << "bye bye!" << endl;
}
```

```
2. #include <iostream>
using namespace std;

int main() {
    int x(7), y = 11;
    char ch = 'B';
    double z(1.34);
    ch += x;
    cout << ch << endl;
    cout << y + z << endl;
    cout << x + y * z << endl;
    cout << x / y * z << endl;
}
```

```
3. #include <iostream>
using namespace std;

bool test(int x, int y)
{ return x / y; }

int main()
{ int m = 17, n = 18;
  cout << test(m,n) << endl;
  cout << test(n,m) << endl;
  m += n;
  n /= 5;
  cout << test(m,n) << endl;
}
```

19.5 Standard Expression Conversions

Expression conversions, including implicit type conversions through promotion or demotion, and explicit casting through a variety of casting mechanisms are discussed in this section.

Suppose x and y are numeric variables. An expression of the form $x \text{ op } y$ has both a value and a type. When this expression is evaluated, temporary copies of x and y are used. If x and y have different types, the one with the shorter type may need to be converted (widened) before the operation can be performed. An implicit conversion of a number that preserves its value is called a **promotion**.

Automatic Expression Conversion

Rules for $x \text{ op } y$

1. Any `bool`, `char`, `signed char`, `unsigned char`, `enum`, `short int`, or `unsigned short int` is promoted to `int`. This is called an **integral promotion**.
2. If, after the first step, the expression is of mixed type, then the operand of smaller type is promoted to that of the larger type, and the value of the expression has that type.
3. The hierarchy of types is indicated by the arrows in Figure 19.1.

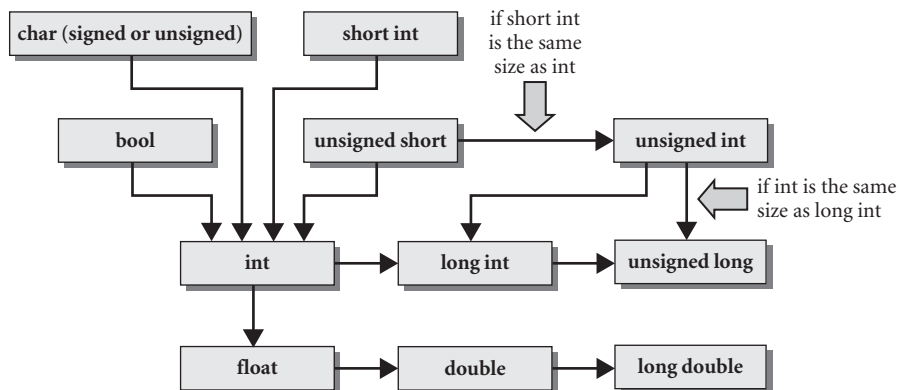


FIGURE 19.1 Hierarchy of basic types

We note that the relationship between `unsigned` and `long` depends on the implementation. For example, on a system that implements `int` with the same number of bytes as `long`, it would not be possible to promote `unsigned` to `long`, so the promotion process would bypass `long` and promote `unsigned` to `unsigned long`. Now assume that we have the following declarations:

```
double d;
int i;
```

In general, a promotion such as `d = i;` will be well behaved. An assignment that causes a demotion such as `i = d;` will result in a loss of information. Assuming the compiler permits the assignment, the fractional part of `d` would be discarded. Example 19.2 demonstrates some of the conversions we have discussed.

EXAMPLE 19.2 `src/mixed-types.cpp`

```
#include <iostream>
using namespace std;

int main() {
    int i, j = 88;
    double d = 12314.8723497;
    cout << "initially d = " << d
         << " and j = " << j << endl;
    cout << "The sum is: " << d + i << endl;
    i = d;
    cout << "after demoting d, i = " << i << endl;
    d = j;
    cout << "after promoting j, d = " << d << endl;
}
```

Here is the compile and run.

```
src> g++ mixed-types.cpp
mixed-types.cpp: In function 'int main()':
mixed-types.cpp:10: warning: converting to 'int' from 'double'
src> ./a.out
initially d = 12314.9 and j = 88
The sum is: 1.34527e+08
after demoting d, i = 12314
after promoting j, d = 88
src>
```

EXERCISE: STANDARD EXPRESSION CONVERSIONS

Assume that we have the following declarations:

```
double d = 123.456;
int i = 789, j = -1;
uint k = 10;
```

- What is the type and what is the value of `d + i`?
- What is the type and what is the value of `j + k`?
- What happens with a promotion such as `d = i`?
- What happens with a demotion such as `i = d`?

19.6 Explicit Conversions

Explicit conversions are called **casts**. Casting is sometimes necessary, but it tends to be overused and can be a major source of errors. In fact, Bjarne Stroustrup, the creator of C++, is on record recommending that they be used as little as possible.

Because of its roots in the C language, C++ supports the old-style (unsafe) C-style casting *(type) expr*:

```
double d=3.14;
int i = (int) d;
```

C++ also supports an alternate constructor-style syntax for casts:

```
Type t = Type(arglist)
```

A cast causes a temporary value of the specified type to be created and pushed onto the program stack. If *Type* is a class, a temporary object is created and initialized by the appropriate conversion constructor. If *Type* is a native type, *Type*(*arg*) is equivalent to *(Type) arg*. The temporary is kept on the stack just long enough to evaluate the expression it is in. After that, it is destroyed.

For example,

```
double d = 3.14;
Complex c = Complex(d);
```

19.7 Safer Typecasting Using ANSI C++ Typecasts

ANSI C++ adds four cast operators, with template-style syntax, that more clearly express the intentions of the programmer and make casts easier to spot in the code. These ANSI typecasts are:

- `static_cast<type>(expr)`—for converting between related types
- `const_cast<type> expr`—for casting away `const` or `volatile`
- `dynamic_cast<type>(expr)`—for safe navigation through an inheritance hierarchy
- `reinterpret_cast<type>(expr)`—for type conversions of pointers, between unrelated types

19.7.1 `static_cast` and `const_cast`

`static_cast<DestType>(expr)` converts the value `expr` to type `DestType`, provided that the compiler knows of an implicit conversion from `expr` to `DestType`. All type-checking is done at compile time.

```
static_cast<char>('A' + 1.0);  
static_cast<double>(static_cast<int>(y) + 1);
```

The `static_cast` operator converts between related types such as one pointer type to another, an enumeration type to an integral type, or a floating-point type to an integral type. These conversions are well defined, portable, and invertible. The compiler can apply some minimal type checking for each `static_cast`.

`static_cast` cannot cast away `constness`. For that you must use `const_cast<DestType>(expr)`, which creates a non-`const` version of `expr`.

In that case, the `DestType` can differ from the type of `expr` only in the presence or absence of `const/volatile`.

For an `int i`, `static_cast<double>(i)` will create a temporary of type `double`, which has the value of `i`. The variable `i` itself is not changed by this cast.

Example 19.3 contains both kinds of casts.

EXAMPLE 19.3 `src/ansicast/m2k.cpp`

```
// Miles are converted to kilometers.  
#include <QTextStream>  
  
QTextStream cin(stdin, QIODevice::ReadOnly);  
QTextStream cout(stdout, QIODevice::WriteOnly);  
QTextStream cerr(stderr, QIODevice::WriteOnly);
```

```

const double m2k = 1.609;    // conversion constant

inline double mi2km(int miles) {
    return (miles * m2k);
}

int main() {
    int miles;
    double kilometers;
    cout << "Enter distance in miles: " << flush;
    cin >> miles ;
    kilometers = mi2km(miles);
    cout << "This is approximately "
         << static_cast<int>(kilometers)
         << "km."<< endl;
    cout << "Without the cast, kilometers = "
         << kilometers << endl;
    double* dp = const_cast<double*>(&m2k);
    cout << "m2k: " << m2k << endl;
    cout << "&m2k: " << &m2k << " dp: " << dp << endl;
    cout << "*dp: " << *dp << endl;
    *dp = 1.892; ❶
    cout << "Can we reach this statement? " << endl;
    return 0;
}

```

Output:

```

Enter distance in miles: 23
This is approximately 37km.
Without the cast, kilometers = 37.007
m2k: 1.609
&m2k: 0x8049048 dp: 0x8049048
*dp: 1.609
Segmentation fault

```

❶ What are we attempting to do here?

Here are some observations regarding the previous example.

- The mixed expression `miles * m2k` is implicitly widened to `double`.
- The safe cast `static_cast<int>(kilometres)` truncates the `double` value to `int`.
- The cast did not change the variable `kilometres`.
- The results of our attempt to assign to `*dp` are undefined.

Casting Away `const`

In general, `const_cast` is only used for `const`-references and pointers to non-`const` objects. Using `const_cast` to change `const` objects has undefined

behavior because `const` objects may be stored in read-only memory (which the operating system protects). In the case of `const int`, trying to change it by casting away `const` depends on compiler optimization techniques, which frequently optimize them out of existence (by doing pre-compilation value replacement). Consider Example 19.4.

EXAMPLE 19.4 `src/casts/constcast1.cpp`

```
#include <iostream>
using namespace std;

int main() {
    const int N = 22;
    int * pN = const_cast<int*>(&N);
    *pN = 33;
    cout << N << '\t' << &N << endl;
    cout << *pN << '\t' << pN << endl;
}
```

Output:

```
22      0xbf91cfa0
33      0xbf91cfa0
```

The above output, obtained with `gcc` version 4.0.3, could be different on your system, because the behavior is undefined.

In this example we used `const_cast` to obtain a regular pointer to a `const int`. Because the `const int` is in stack storage class, we don't get a segmentation fault by attempting to change the memory. The compiler is unable to "optimize out" the `int`, and the `const_cast` tells it not to even try.

EXERCISES: STATIC_CAST AND CONST_CAST

- In Example 19.4, try moving the `const int N = 22;` above or below `int main() {`. Observe and explain the difference in output.
- Predict the output of Example 19.5. Remove the `const_cast` from the call to `f2()` inside `f1()`, and predict the output again.

EXAMPLE 19.5 `src/casts/constcast2.cpp`

```
#include <iostream>

void f2(int& n) {
    ++n;
}

void f1(const int& n, int m) {
    if(n < m)
        f2(const_cast<int&>(n));
}

using namespace std;

int main() {
    int num1(10), num2(20);
    f1(num1, num2);
    cout << num1 << endl;
}
```

19.7.2 `reinterpret_cast`

`reinterpret_cast` is used for casts that are representation- or system-dependent; examples are conversions between unrelated types such as `int` to pointer or between unrelated pointer types such as `int*` to `double*`. It cannot cast away `const`. `reinterpret_casts` are dangerous, generally not portable, and should be avoided.

Consider the following situation.

```
Spam spam;
Egg* eggP;
eggP = reinterpret_cast<Egg*>(&spam);
eggP->scramble();
```

`reinterpret_cast` takes some `spam` and gives us an Egg-shaped pointer, without any concern for type compatibility.

By using `eggP`, we are *reinterpreting* the bits of `spam` as if they were bits of `egg`. In some countries, this would be sacrilege!

What Is It Really Used For?

Sometimes, a C function returns a `void*` pointing to a type that is known to the developer. In such a case, a typecast from `void*` to the actual type is needed. If you are sure it is pointing to an `Egg`, `reinterpret_cast<Egg*>` is the appropriate cast to use. There is no compiler or runtime checking on such a cast.

19.7.3 Why Not Use C-style Casts?

C-style casts are deprecated and should not be used anymore. Consider the following situation, quite similar to the previous example.

```
Apple apple;
Orange* orangeP;
// other processing steps ...
orangeP = (Orange*) &apple;
orangeP->squeeze();
```

The problem is that we can not tell from looking at this code whether the developer is aware that an `Apple` is not compatible with an `Orange`. From looking at it, it is unclear whether this is a proper type conversion or a non-portable pointer conversion.

Errors caused by such a cast can be very difficult to understand and correct. If a system-dependent cast is necessary, it is preferable to use `reinterpret_cast` over a C-style cast so that, when troubles arise, it will be easier to spot the likely source of those troubles in the code.

19.8 Run-Time Type Identification (RTTI)

This section covers `dynamic_cast` and `typeid`, two operators that enable runtime type identification (RTTI).

When operating on hierarchies of types, sometimes it is necessary to downcast a pointer to a more specific type. Without a downcast, only the interface of the pointer type (the base class) is available. One common situation where downcasting is used is inside functions that accept base class pointers.

The conversion of a base class pointer to a derived class pointer is called **downcasting** because casting from the base class to a derived class is considered moving *down* the class hierarchy.

RTTI allows programmers to safely convert pointers and references to objects from base to derived types.

`dynamic_cast<D*>(ptr)` takes two operands: a pointer type `D*` and a pointer `ptr` of a polymorphic type `B*`. If `D` is a base class of `B` (or if `B` is the same as `D`), `dynamic_cast<D*>(ptr)` is an upcast (or not a cast at all) and is equivalent

to `static_cast<D*>(ptr)`. But if `ptr` has the address of an object of type `D`, where `D` is derived from `B`, the operator returns a downcast pointer of type `D*`, pointing to the same object. If the cast is not possible, then a null pointer is returned.

`dynamic_cast` performs runtime checking to determine whether the pointer/reference conversion is valid.

Example 19.6 shows operations on a collection of `QWidget`s. In fact, we wish to operate only buttons and sliders, leaving the rest alone.

EXAMPLE 19.6 `src/rtti/dynamic_cast.cpp`

```
[ . . . . ]
int processWidget(QWidget* wid) {

    if (widpointer->inherits("QAbstractSpinBox")) { ❶
        QAbstractSpinBox* qasbp =
            static_cast <QAbstractSpinBox*> (widpointer);
        qasbp->setAlignment(Qt::AlignHCenter);
    }
    else {
        QAbstractButton* buttonPtr =
            dynamic_cast<QAbstractButton*>(widpointer);
        if (buttonPtr) { ❷
            buttonPtr->click();
            qDebug() << QString("I clicked on the %1 button:")
                .arg(buttonPtr->text());
        }
        return 1;
    }
    return 0;
}
[ . . . . ]
QVector<QWidget*> widvect;

widvect.append(new QPushButton("Ok"));
widvect.append(new QCheckBox("Checked"));
widvect.append(new QComboBox());
widvect.append(new QMenuBar());
widvect.append(new QCheckBox("With Fries"));
widvect.append(new QPushButton("Nooo!!!!"));
widvect.append(new QDateTimeEdit());
widvect.append(new QDoubleSpinBox());
foreach (QWidget* widpointer, widvect) {
    processWidget(widpointer);
}
return 0;
}
```

❶ only for QObjects processed by moc

❷ If non-null, it's a valid QAbstractButton.



`qobject_cast` (see Section 15.3) is faster than `dynamic_cast`, but only works on `QObject`-derived types.

In terms of run-time cost, `dynamic_cast` is considerably more expensive, perhaps 10 to 50 times the cost of a `static_cast`. However, they are not interchangeable operations and are used in very different situations.

19.8.1 typeid Operator

Another operator that is part of RTTI is `typeid()`, which returns type information about its argument. For example:

```
void f(Person& pRef) {
    if(typeid(pRef) == typeid(Student) {
        // pRef is actually a reference to a Student object.
        // Proceed with Student specific processing.
    }
    else {
        // The Object referred to by pRef is not a Student.
        // Do whatever alternative stuff is required.
    }
}
```

`typeid()` returns a `type_info` object that corresponds to the argument's type.

If two objects are the same type, their `type_info` objects should be equal. The `typeid()` operator can be used for polymorphic types or non-polymorphic types. It can also be used on basic types as well as custom classes. Furthermore, the arguments to `typeid()` can be type names or object names.

This is one possible implementation of the `type_info` class.

```
class type_info {
private:
    type_info(const type_info& );
    // cannot be copied by users
    type_info& operator=(const type_info&);
    // implementation dependent representation
public:
    virtual ~type_info();
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
    // returns a pointer to the name of the type
}
```

19.9 Member Selection Operators

There are two forms of the member selection operator:

1. *pointer->memberName*
2. *object->memberName*

They look the same, but differ in the following ways.

1. The first is binary, the second is unary.
2. The first is global and not overloadable, the second is an overloadable member function.

When it is defined for a class, the unary operator `->()` should return a pointer to an object that has a member whose name is *memberName*.

An object that implements `operator->` is typically called a **smart pointer**. Smart pointers are so called because they can be programmed to be “smarter” than ordinary pointers. For example, a `QPointer` is automatically set to 0 when the referenced object is destroyed. Examples of smart pointers include

1. STL style iterators
2. `auto_ptr`—the STL smart pointer
3. `QPointer`—the Qt 4 smart pointer

All of these smart pointers are template classes.

Example 19.7 shows the source code for `QPointer`.

EXAMPLE 19.7 `src/pointers/autoptr/qpointer.h`

```
[ . . . . ]
template <class T>
class QPointer
{
    QObject *o;
public:
    inline QPointer() : o(0) {}
    inline QPointer(T *p) : o(p)
        { QMetaObject::addGuard(&o); }
    inline QPointer(const QPointer<T> &p) : o(p.o)
        { QMetaObject::addGuard(&o); }
    inline ~QPointer()
        { QMetaObject::removeGuard(&o); }
    inline QPointer<T> &operator=(const QPointer<T> &p)
        { if (this != &p) QMetaObject::changeGuard(&o, p.o); return
            *this; }
    inline QPointer<T> &operator=(T* p)
        { if (o != p) QMetaObject::changeGuard(&o, p); return *this; }
```

continued

```

inline bool isNull() const
    { return !o; }

inline T* operator->() const
    { return static_cast<T*>(const_cast<QObject*>(o)); }
[ . . . . ]

```

The operators make use of two ANSI-style typecasts (Section 19.7): `static_cast` and `const_cast`. `const_cast` is required because the function is `const`, and it must return a non-`const` pointer. RTTI is not necessary here because the template ensures that the actual pointer stored was a `T*`. Therefore, `static_cast` can be used instead of `dynamic_cast`.

A `QPointer<T>` is said to be a *guarded* pointer to a `QObject` of type `T`. By guarded we mean that the `QPointer` is automatically set to 0 if the object that it is pointing to is destroyed. Here is a code fragment that demonstrates how a `QPointer` can be used.

```

[. . .]
QPointer<QIntValidator> val = new QIntValidator(someParent);
val->setRange(20, 60);
[. . .]

```

In that second line of code, `val->` returns a pointer to the newly allocated object, and that pointer is used to access the `setRange()` member function.

POINT OF DEPARTURE

Guarded pointers are discussed further in *Qt Quarterly*.¹

EXERCISES: TYPES AND EXPRESSIONS

1. Imagine you are required to use a library of classes that have been poorly written and you have no way to improve them. (It could happen!) The code in Example 19.8 includes one such badly written example class and a small program that uses it. The program logic shows some objects being created and passed to a function that receives them as `const` references (an implicit vow not to change them) and then prints an arithmetic result. Unfortunately, because the class was badly written, something goes wrong along the way.

¹ <http://doc.trolltech.com/qq/qq14-guardedpointers.html>

EXAMPLE 19.8 `src/const/cast/const.cc`

```
#include <iostream>
using namespace std;

class Snafu {
public:
    Snafu(int x) : mData(x) {}
    void showSum(Snafu & other) const {
        cout << mData + other.mData << endl;
    }

private:
    int mData;
};

void foo(const Snafu & myObject1,
         const Snafu & myObject2) {
    // [ . . . ]
    myObject1.showSum(myObject2);
}

int main() {

    Snafu myObject1(12345);
    Snafu myObject2(54321);

    foo(myObject1, myObject2);

}
```

Answer these questions.

- a.** What went wrong?
- b.** What change to the class would fix it?

Unfortunately, you can't change the class. Come up with at least two ways to fix the program without changing the class definition. What would be the best of these and why?

- 2.** The code in Example 19.9 is an incomplete attempt to create a class that counts, for each instantiation, the number of times an object's data are printed. Review the program and then make it work properly.

EXAMPLE 19.9 src/const/cast/const2.cc

```
#include <iostream>
using namespace std;

class Quux {
public:
    Quux(int initializer) :
        mData(initializer), printcounter(0) {}
    void print() const;
    void showprintcounter() const {
        cout << printcounter << endl;
    }

private:
    int mData;
    int printcounter;
};

void Quux::print() const {
    cout << mData << endl;
}

int main() {
    Quux a(45);
    a.print();
    a.print();
    a.print();
    a.showprintcounter();
    const Quux b(246);
    b.print();
    b.print();
    b.showprintcounter();
    return 0;
}
```

REVIEW QUESTIONS

1. What is the difference between a statement and an expression?
2. What is the difference between an overloaded operator and a function?
3. What ways can you introduce a new type into C++?
4. Which cast operator is best suited for numeric values?
5. What happens when you assign an `int` variable to a `double` value?
6. Which cast operator is best suited for downcasting through polymorphic hierarchies?
7. Why are ANSI casts preferred over C-style casts?
8. What is a situation in which you might find the `reinterpret_cast` used in a reasonable way?

20

CHAPTER 20

Scope and Storage Class

Identifiers have scope, objects have a storage class, and variables have both. In this chapter, we discuss the difference between declarations and definitions, and how to determine the scope of identifiers and the storage class of objects.

20.1	Declarations and Definitions	464
20.2	Identifier Scope	465
20.3	Storage Class	470
20.4	Namespaces	473

20.1 Declarations and Definitions

Any identifier must be declared or defined before it is used.

Declaring a name means telling the compiler what type to associate with that name. The location of the declaration determines its **scope**. Scope is a region of code where an identifier can be used.

Defining an object, or variable, means allocating space and (optionally) assigning an initial value. For example,

```
double x, y, z;
char* p;
int i = 0;
QString message("Hello");
```

Defining a function means completely describing its behavior in a block of C++ statements. For example,

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

Defining a class means specifying its structure in a sequence of declarations of function and data members.

EXAMPLE 20.1 `src/early-examples/decldef/point.h`

```
class Point { ❶
public:
    Point(int x, int y, int z); ❷
    int distance(Point other); ❸
    double norm() const { ❹
        return distance(Point(0,0,0));
    }
private:
    int m_Xcoord, m_Ycoord, m_Zcoord; ❺
};
```

- ❶ a class definition
 - ❷ a constructor declaration
 - ❸ a function declaration
 - ❹ declaration and definition
 - ❺ data member declaration
-

Example 20.2 contains some declarations that are not definitions.

EXAMPLE 20.2 `src/early-examples/decldef/point.cpp`

```
extern int step;           ❶  
class Map;                ❷  
int max(int a, int b);    ❸
```

- ❶ an object (variable) declaration
- ❷ a class declaration
- ❸ a global (non-member) function declaration

In each case, there is an implicit promise to the compiler (which will be enforced by the linker) that the declared name will be defined somewhere else in the program.

Each definition *is* a declaration. There can be only one definition of any name in any scope, but there can be multiple declarations.



Variable initialization is optional in C++. Nevertheless, it is strongly recommended that an initial value be provided for all variable definitions, otherwise invalid results or strange runtime errors can occur that are often difficult to locate. It is worth repeating this rule: *All objects and variables should be properly initialized at creation time.*

20.2 Identifier Scope

Every identifier has a scope that is determined by where it is declared. **Identifier Scope** in a program is the region(s) of the program within which the identifier can be used. Using a name outside of its scope is an error.

The same name may declared/used in different scopes. Ambiguities are resolved as follows:

1. The name from the most local scope is used.
2. If the name is not defined in the most local scope, the same name defined in the nearest enclosing scope will be used.
3. If the name is not defined in any enclosing scope, then the compiler will report an error.

There are five possible scopes in C++.

1. Block scope (local to a block of statements)
2. Function scope (the entire extent of a function)¹

¹ Only labels have function scope.

3. Class scope (the entire extent of a class, including its member functions)
4. File scope (the entire extent of a source code file)
5. Global scope (the entire extent of a program)

Because the compiler only deals with one source file at a time, only the Linker can tell the difference between global and file scope, as Example 20.3 shows.

EXAMPLE 20.3 Global versus File scope

```
// In File 1:
int g1;           // global
int g2;           // global
static int g3;   // keyword static limits g3 to file scope
(etc.)

// In File 2:
int g1;           // linker error!
extern int g2;    // okay, share variable space
static int g3;   // okay: 2 different variable spaces
(etc.)
```

20.2.1 Default Scope of Identifiers: A Summary

Let's look at the five scopes in a bit more detail.

1. **Block scope.** An identifier declared inside curly braces (excluding namespace blocks) `{ ... }` or in a function parameter list has block scope. Block scope extends from the declaration to the enclosing right brace.
2. **Function scope.** Labels in C/C++ have their own scope. They are accessible before and after their declaration, for the whole function. C supports a very rarely used and often shunned `goto` statement that requires a label. The thing that makes its scope unique is that the label (the declaration) can appear *after* the first `goto` that uses it. Example 20.4 shows an example of its use. `goto` is seldom used by serious programmers but labels, because they exist, sometimes pop up in code and are used to solve various compatibility problems.²



Avoid `goto` and labels in your code.

² Such as preventing the C++ compiler from choking on the `signals:` and `slots:` declarations.

3. **Class scope.** An identifier that is declared inside a class definition has **class scope**. Class scope is anywhere in the class definition³ or in the bodies of member functions.⁴
4. **File scope.** This is basically the same as global scope, except that file scope variables are not exported to the linker. The keyword `static` *hides* an identifier from other source files and gives it file scope. File scope variables cannot be declared `extern` and accessed from another file.

File scope variables, because they are not exported, do not pollute the global namespace. They are often used in C programs because C has no concept of `private` class members.



File scope is available in C++ for backward compatibility with C, but we prefer to use namespaces or `static` class members instead.

5. **Global scope.** An identifier whose declaration is not in between braces (round or curly) has global scope. The scope of such an identifier begins at the declaration and extends from there to the bottom of the source code file. `extern` declarations may be used to access global definitions in other source files.

An identifier in a `namespace` is available from other scopes through the `using` keyword. It is also available globally through the use of the scope resolution `::` operator. Namespace variables and static class members are accessible globally and have static storage, like global variables, except that they do not enlarge the global namespace. See Section 20.4 for more details.



Use of global scope for variables is unnecessary in C++. In general, only classes and namespaces should be defined in global scope. If you need a “global” variable, you can achieve something similar through the use of `public static` class member or a namespace member.

EXAMPLE 20.4 `src/goto/goto.cpp`

```
[ . . . . ]
int look() {
    int i=0;
    for (i=0; i<10; ++i) {
```

continued

³ Including inline function definitions above the declarations of referred members

⁴ Keeping in mind that the scope of non-`static` members *excludes* the bodies of `static` member functions

```

        if (i == rand() % 20)
            goto found; ❶
    }
    return -1;
found: ❷
    return i;
}
[ . . . . ]

```

❶ It would be better to use `break` or `continue`.

❷ `goto` serves as a forward declaration to a label.

20.2.2 File Scope versus Block Scope and operator::

We have seen and used the scope resolution operator to extend the scope of a class or access its members with `ClassName::`. A similar syntax is used to access the individual symbols in a namespace with `NamespaceName::`. C++ also has a (unary) file scope resolution operator, `::`, that provides access to global, namespace, or file scope objects from inside an enclosed scope. The following exercise deals with the use of this operator with various scopes.

EXERCISE: FILE SCOPE VERSUS BLOCK SCOPE AND OPERATOR::

Determine the scope of each of the variables in Example 20.5. Then be the computer and predict the output of the program.

EXAMPLE 20.5 `src/early-examples/scopex.cpp`

```

#include <iostream>
using namespace std;

long x = 17;
float y = 7.3; ❶
static int z = 11; ❷

class Thing {
    int m_Num; ❸
public:
    static int sm_Count; ❹
    Thing(int n = 0) : m_Num(n) { ++sm_Count; }
    ~Thing() { --sm_Count; }
    int getNum() { return m_Num; }
};

```



```

int Thing::sm_Count = 0;
Thing t(11);

int fn(char c, int x) {
    int z = 5;
    double y = 6.933;
    {
        char y;
        Thing z(4);
        y = c + 3;
        ::y += 0.3;
        cout << y << endl;
    }
    cout << Thing::sm_Count
        << endl;
    y /= 3.0;
    ::z++;
    cout << y << endl;
    return x + z;
}

int main() {
    int x, y = 10;
    char ch = 'B';
    x = fn(ch, y);
    cout << x << endl;
    cout << ::y << endl;
    cout << ::x / 2 << endl;
    cout << ::z << endl;
}

```

- 1 Scope: _____
 - 2 Scope: _____
 - 3 Scope: _____
 - 4 Scope: _____
 - 5 Scope: _____
 - 6 Scope: _____
 - 7 Scope: _____
 - 8 Scope: _____
 - 9 Scope: _____
 - 10 Scope: _____
 - 11 Scope: _____
 - 12 Scope: _____
 - 13 Scope: _____
 - 14 Scope: _____
 - 15 Scope: _____
-

20.3 Storage Class

Whenever an object is created, space is allocated in one of four possible places. Each of these places is called a **storage class**.

While scope refers to a region of code where an identifier is accessible, storage class refers to a location in memory.

1. The **static** area: Global variables, `static` locals, and `static` data members are all stored in the `static` storage area. The lifetime of a static object begins when its object module loads and ends when the program terminates.

It is used often for pointers, simple types, and string constants, less often for complex objects.

2. The program **stack** (automatic storage—`auto`⁵): Local variables, function parameters, and temporary objects are all stored on the stack. For local (block-scope) variables, the lifetime is determined by the brackets around the code that is executed.

Objects on the stack include function parameters, return values, local or temporary variables. Stack storage is allocated automatically when an object definition is executed. Objects in this storage class are local to a function or a block of statements.⁶

3. The **heap** or free storage (dynamic storage): Objects created via `new` are examples.

The lifetime of a heap object is determined entirely by the use of `new` and `delete`.

In general, the allocation and freeing of heap objects should be in carefully encapsulated classes.

4. Another storage class, left over from C, is called **register**. It is a specialized form of automatic storage. This category of storage can be requested by using the keyword, `register` in the variable declaration. Most C++ compilers ignore this keyword and put such variables on the stack. Using this storage class for an object means that you cannot take its address.

⁵ The optional keyword `auto` is almost never used.

⁶ Or to a member of another object that is

20.3.1 Globals, statics, and QObject

Global objects need to be “global” for two reasons.

1. They need a lifetime that is the same as the application.
2. They need to be available from many different places in the application.

In C++, we avoid the use of global scope at all costs, in favor of other approaches. We still use global scope identifiers for the following:

1. Class names
2. Namespace names
3. The global pointer `qApp`, which points to the running `QApplication` object

By turning a global object into a `static` class member, or a namespace member, we can avoid increasing the size of the global namespace while keeping the object accessible from other source code modules.

statics and QObject

When creating `QObject` or other interesting classes,⁷ it is important that their destructors do not get called after the `QApplication` has been destroyed (after `main()` is finished).

`static` `QObject`s (and other complex objects) cleaned up after a `QApplication` has been destroyed could have difficulties in clean-up code. This is because `QApplication`, when it is destroyed, takes a lot of other objects with it.

In general, we want to have control over the order of destruction of all complex objects. One way to ensure this is to make each `QObject` allocated on the stack, or on the heap, and then added as a child/grandchild of another `QObject` already on the stack.

The `QApplication` (or its derived instance) is the “rootiest” stack object of them all, so we try to make it the “last `QObject` standing.”

20.3.1.1 Globals and const

The scope of `const` variables is slightly different from the scope of regular globals.

A global object that has been declared `const` has file scope, by default. It may be exported to other files by declaring it `extern` at the point where it is initialized.

⁷ By “interesting” we mean any class with a destructor that has some important cleaning up to do.

For example, in one file, we could have this:

```
const int NN = 10;          //must be initialized

//declaration and definition - allocates storage
extern const int QQ = 7;

int main() {
    NN = 12;               // Error!
    int array[NN];        // OK
    QQ++;                 // Error!
    return 0;
}
```

In another file, we might have

```
const int NN = 33;        // a different constant

// declare global constant - storage allocated elsewhere
extern const int QQ;     // external declaration
...
```

The second file has a `const int NN` that is separate and distinct from the `const` with the same name in the first file. The second file can share the use of the `const int QQ` because of the `extern` modifier.

EXERCISE: STORAGE CLASS

In Example 20.6, identify the scope/storage class of each of object's creation/definition. If there is a name clash, indicate there is an error with the definition.

EXAMPLE 20.6 `src/storage/storage.cpp`

```
#include <qstd.h>
using namespace qstd;

int i;                                ❶
static int j;                          ❷
extern int k;                           ❸
const int l=10;                         ❹
extern const int m=20;                  ❺

class Point                             ❻
{
    public:
        QString name;                   ❼
        QString toString() const;
    private:
        static int count;
        int x, y;                       ❽
};

int Point::count = 0;                   ❾
```

```

QString Point::toString() const {
    return QString("%1,%2").arg(x).arg(y); ❿
}

int main(int argc, char** argv)           ❶
{
    int j;                                 ❷
    register int d;                         ❸
    int* ip = 0;                            ❹
    ip = new int(4);                        ❺
    Point p;                                ❻
    Point* p2 = new Point();                ❼
}

```

- ❶ Scope: _____ Storage class: _____
- ❷ Scope: _____ Storage class: _____
- ❸ Scope: _____ Storage class: _____
- ❹ Scope: _____ Storage class: _____
- ❺ Scope: _____ Storage class: _____
- ❻ Scope: _____ Storage class: _____
- ❼ Scope: _____ Storage class: _____
- ❽ Scope: _____ Storage class: _____
- ❾ Scope: _____ Storage class: _____
- ❿ Scope: _____ Storage class: _____
- ⓫ S/SC of argc and argv: _____
- ⓬ Scope: _____ Storage class: _____
- ⓭ Scope: _____ Storage class: _____
- ⓮ Scope: _____ Storage class: _____
- ⓯ Scope: _____ Storage class: _____

20.4 Namespaces

In C and C++ there is one global scope that contains

- The names of all global functions and variables
- Class and type names that are commonly available to all programs

Classes are one way of grouping names (members) under a common heading (the classname), but sometimes it is desirable to have a higher level grouping of names.

The **namespace** mechanism provides a way to partition the global scope into individually named sub-scopes. This helps avoid naming conflicts that can arise when developing a program that uses modules with name conflicts.

The syntax for defining a namespace is

```
namespace namespaceName { decl1, decl2, ... }
```

Any legal identifier can be used for the optional *namespaceName*.

Examples 20.7 and 20.8 define two separate namespaces in different files, each containing functions with the same name.

EXAMPLE 20.7 `src/namespace/a.h`

```
#include <iostream>
namespace A {
    using namespace std;
    void f() {
        cout << "f from A\n";
    }

    void g() {
        cout << "g from A\n";
    }
}
```

EXAMPLE 20.8 `src/namespace/b.h`

```
#include <iostream>

namespace B {
    using namespace std;
    void f() {
        cout << "f from B\n";
    }

    void g() {
        cout << "g from B\n";
    }
}
```

Example 20.9 includes both header files and uses scope resolution to call functions declared in either file.

EXAMPLE 20.9 `src/namespace/namespace1.cc`

```
#include "a.h"
#include "b.h"

int main() {
    A::f();
    B::g();
}
```

Output:

```
f from A
g from B
```

The *using* keyword permits individual members of a namespace to be referenced without scoping. The syntax can take two forms.

1. The **using directive**: `using namespace namespaceName`—imports the entire namespace into the current scope
2. The **using declaration**: `using namespaceName::identifier`—imports a particular identifier from that namespace

Care must be exercised to make sure that ambiguities are not produced when identifiers are present in more than one included namespace. We show an example of such an ambiguous function call in Example 20.10.

EXAMPLE 20.10 `src/namespace/namespace2.cc`

```
#include "a.h"
#include "b.h"

int main() {
    using A::f;           ❶
    f();
    using namespace B;   ❷
    g();                 ❸
    f();                 ❹
}
```

Output:

```
f from A
g from B
f from A
```

- ❶ declaration—brings `A::f()` into scope
 - ❷ directive—brings all of `B` into scope
 - ❸ okay
 - ❹ ambiguous
-



NAMESPACE ALIASES To make the names of various namespaces unique, programmers sometimes produce extremely long namespace names. One can easily introduce an alias for a long namespace name with a command such as:

```
namespace xyz = verylongcomplicatednamespace;
```

20.4.1 Anonymous Namespaces

A namespace without a name is an anonymous namespace. This is similar to `static global`, or file scope identifier. It is accessible from that point down to the end of the file.⁸

Example 20.11 shows how anonymous namespaces can eliminate the need for `static` globals.

EXAMPLE 20.11 `src/namespace/anonymous.h`

```
namespace {
    const int MAXSIZE = 256;
}

void f1() {
    int s[MAXSIZE];
}
```

20.4.2 Open Namespaces

Any namespace definition is **open** in the sense that you can add members to an existing namespace by declaring a second namespace with the same name but with new items. The new items will be appended to the namespace in the order in which the namespace declarations are encountered by the compiler.

Classes are similar to namespaces but classes are *not open* because they serve as a pattern for the creation of objects.

The `using` directive does not extend the scope in which it is used; it imports names from the specified namespace into the current scope.

Names locally defined take precedence over names from the namespace (which are still accessible using the scope resolution operator).

20.4.3 Namespace, static Objects and `extern`

Objects declared inside namespaces are implicitly `static`, meaning that they are created once for the entire application. The initialization of a static object must exist in only one C++ module. To declare a `static` object (global or namespace) without defining it, use the keyword `extern`.⁹ Example 20.12 shows how to declare namespace variables.

⁸ Unless it appears inside another namespace (which the language permits), in which case the scope is further narrowed by the brackets of its enclosing namespace.

⁹ Even inside namespaces!

EXAMPLE 20.12 `src/libs/utls/qstd.h`

```
[ . . . . ]
namespace qstd {
    extern QTextStream cin;      ❶
    extern QTextStream cout;
    extern QTextStream cerr;
    bool yes(QString yesNoQuestion);
    bool more(QString prompt);
    int promptInt(int base = 10);
    double promptDouble();
    void promptOutputFile(QFile& outfile);
    void promptInputFile(QFile& infile);

```

```
[ . . . . ]
```

❶ declared only—defined in the .cpp file

Each variable must be defined in a .cpp file, as shown in Example 20.13.

EXAMPLE 20.13 `src/libs/utls/qstd.cpp`

```
[ . . . . ]
QTextStream qstd::cin(stdin, QIODevice::ReadOnly);
QTextStream qstd::cout(stdout, QIODevice::WriteOnly);
QTextStream qstd::cerr(stderr, QIODevice::WriteOnly);

```

REVIEW QUESTIONS

1. What is a scope? What kinds of “things” have a scope?
2. What is a storage class? What kinds of “things” have a storage class?
3. When are `static` objects initialized? Be sure to mention both globals and block-scope objects.
4. How does `const` act as a scope modifier?
5. What does `extern` mean?
6. The keyword `static` has many meanings depending upon where it is used.
 - a. Explain how `static` can be used as a scope modifier.
 - b. Explain how `static` can be used as a storage-class modifier.
 - c. Give another use for the keyword `static`.

21

CHAPTER 21

Statements and Control Structures

Programs consist of statements of various kinds. Control structures are statements that control the way other statements are executed. This chapter formally defines the language elements and shows what kind of control structures are available.

21.1 Statements	480
21.2 Selection Statements	480
21.3 Iteration	483
21.4 Exceptions	485

21.1 Statements

A C++ program contains **statements** that alter the state of the storage managed by the program and determine the flow of program execution. There are several types of C++ statements, most of which are inherited from the C language. First, there is the simple statement, terminated with a semicolon, such as

```
x = y + z;
```

Next, there is the **compound statement**, or **block**, consisting of a sequence of statements enclosed in curly braces.

```
{
    int temp = x;
    x = y;
    y = temp;
}
```

The above example is a single compound statement that *contains* three simple statements. The variable `temp` is local to the block and is destroyed when the end of the block is reached.

In general, a compound statement can be placed wherever a simple statement can go. The reverse is not always true, however. In particular, the function definition

```
double area(double length, double width) {
    return length * width;
}
```

cannot be replaced by

```
double area(double length, double width)
    return length * width;
```

The body of a function definition must always include a block.

21.2 Selection Statements

Every programming language has at least one control structure that allows the flow of the program to branch depending on the outcome of a boolean condition.

In C and C++ we have `if` and `switch`. The `if` statement typically has the following form.

```
if(boolExpression)
    statement
```

It can have an optional `else` attached.

```
if(boolExpression)
    statement1
else
    statement2
```

Conditional statements can be nested, which means that they can get quite complicated. A very important rule to keep in mind is that an `else` or `else if` clause is activated when the *boolExpression* of the *immediately preceding* open `if` evaluates to `false`. This can be confusing when your program logic allows you to omit some `else` clauses. Consider the following badly indented example, where `x` is an `int`.

```
if (x>0)
    if (x > 100)
        cout << "x is over a hundred";
else
    if (x == 0) // no! this cannot be true -the indentation is misleading
        cout << "x is 0";
    else
        cout << "x is negative"; // no! x is between 1 and 100 inclusive!
```

We can clarify and repair this logic with braces.

```
if (x>0) {
    if (x > 100)
        cout << "x is over a hundred";
}
else
    if (x == 0) // now this is possible.
        cout << "x is 0";
    else
        cout << "x is negative";
```

An `if` without an `else` can be closed by enclosing the `if` statement in braces `{}`, making it a compound statement.

`switch` is another branching construct, which permits the execution of different code depending the value of a parameter.

```
switch(integralExpression) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    ...
```

continued

```
    case valuen:
        statementn;
        break;
    default:
        defaultStatement;
}
nextStatement;
```

The `switch` statement is a computed `goto` statement. Each case is followed by a unique case label value, which is compared to the *integralExpression*.

When the `switch` causes a jump to a case label whose value matches the *integralExpression*, statements are executed from that point on until the end of the `switch` block or a branching statement (e.g., `break`) is reached.

The optional `default` label is the jump destination when the *integralExpression* does not match any case label value. If `default` is omitted, and no matching case label exists, then the `switch` block is skipped.

The *integralExpression* must be an expression that evaluates to an integer. Each case label, except `default`, must be an integer constant.¹

Any `switch` statement such as the one above can be rewritten as a long `if ... else` statement. However, the runtime performance of a `switch` is considerably better because it requires only a single comparison and performs only one branch.

```
if(integralExpression == value1)
    statement1;
else if(integralExpression == value2)
    statement2;
...
else if(integralExpression == valuen)
    statementn;
else
    defaultStatement;
```



Long compound conditional statements and `switch` statements should be avoided in object-oriented programming (unless they are isolated in factory code), because they tend to make functions complex and hard to maintain.

If each case can be rewritten as a method of a different class, you can use the Strategy pattern (and the virtual table) instead of writing your own `switch` statement.

¹ case labels are not the same as `goto` labels that are used as destinations for the infamous `goto` statement. `goto` labels must be identifiers. In particular, they cannot be integers.

EXERCISE: SELECTION STATEMENTS

Be the computer and predict the output of the program in Exercise 21.1. Then run it and compare your predicted output with the output produced by the computer.

EXAMPLE 21.1 `src/early-examples/nestedif.cpp`

```
#include <iostream>
using namespace std;

void nestedif1 () {
    int m = 5, n = 8, p = 11;
    if (m > n)
        if (p > n)
            cout << "red" << endl;
        else
            cout << "blue" << endl;
}

void nestedif2() {
    int m = 5, n = 8, p = 11;
    if (m > n) {
        if (p > n)
            cout << "red" << endl;
    } else
        cout << "blue" << endl;
}

int main() {
    nestedif1();
    nestedif2();
    return 0;
}
```

21.3 Iteration

C++ provides three iteration structures.

1. `while`

```
while ( loopCondition ) {
    loopBody
}
```

- a. Evaluate *loopCondition* first.
- b. Execute *loopBody* repeatedly until *loopCondition* is false.

2. `do..while:`

```
do {
    loopBody
} while ( loopCondition ) ;
```

 - a. Execute *loopBody* first.
 - b. Evaluate *loopCondition*.
 - c. Execute *loopBody* repeatedly until *loopCondition* is *false*.
3. `for loop:`

```
for ( initStatement; loopCondition; incrStmt ) {
    loopBody
}
```

 - a. Execute *initStatement* first.
 - b. Execute *loopBody* repeatedly until *loopCondition* is *false*.
 - c. After each execution of *loopBody*, execute *incrStmt*.

With each of these iteration structures, the *loopBody* code gets repeated as long as *loopCondition* evaluates to boolean `true`. The `do` loop differs from the other two in that its *loopCondition* gets checked at the bottom of the loop, so its *loopBody* is always executed at least once.

A common programming error is to place a semicolon after the `while`.

```
while (notFinished());
    doSomething();
```

The first semicolon terminates the `while` statement entirely and produces a loop with an empty *loopBody*. Even though `doSomething()` is indented, it does not get executed inside the loop. The *loopBody* is responsible for changing the *loopCondition*. If `notFinished()` is initially `true`, then the empty *loopBody* will cause an infinite loop. If `notFinished()` is initially `false`, then the loop terminates immediately and `doSomething()` gets executed exactly once.

C++ provides `break` and `continue` for finer control over code executed inside loops.

```
while ( moreWorkToDo ) {
    statement1;
    if ( specialCase ) continue;
    statement2;
    if ( noMoreInput ) break;
    statement3;
// continue jumps here
}
// break jumps here
```

- `break` jumps out of the current control structure, whether it is a `switch`, `for`, `while`, or `do..while`.
- `continue` only operates inside loops, and it skips the rest of the current iteration to the check if there is *moreWorkToDo*.

EXERCISES: ITERATION

1. Write the function `isPrime(int n)` that returns true if `n` is prime and false otherwise. Supply an interactive `main()` to test your function.
2. Write the function `primesBetween(int min, int max)` that displays on the screen all the prime numbers that are between `min` and `max`. Supply an interactive `main()` to test your function.
3. Write the function `power2(int n)` that computes and returns the value of 2 raised to the power `n`. Supply an interactive `main()` to test your function.
4. Write a binary logarithm function `binLog(int n)` that computes and returns an `int` equal to the integral part of $\log_2(n)$, where `n` is positive. This is equivalent to finding the exponent of the largest power of 2 that is less than or equal to `n`. For example, `binLog(25)` is 4. There are at least two simple, iterative ways to do this computation. Supply an interactive `main()` to test your function.

21.4 Exceptions

We all know that things sometimes break and that improbable or unforeseen events can mess up the most carefully laid plans. This is especially true with programming and might explain why programmers have such great respect for Murphy's Law.² When we write programs we try to anticipate the things that can go wrong and make allowances for them, so we try to make our programs foolproof. This may seem inconsistent with a very important corollary to Murphy's law,³ but we persevere anyway.

Exception handling permits the program to respond to an exceptional situation by invoking a procedure that is specifically designed to handle it. This can allow the program to recover from a condition that might normally cause it to crash or get into a bad state.

Exception handling was added to C++ relatively late in the development of the language. As with any good thing, there is a cost associated with exception handling. Enabling exception handling adds a significant amount of additional code to the executable, which can degrade run-time performance. Some developers avoid this feature completely. Qt 4 library code, for example, does not require any exception handling by client code.

² Whatever can go wrong will go wrong.

³ It is impossible to make anything foolproof because fools are too clever.

21.4.1 Exception Handling

Here is a summary of how exception handling works in C++. An **exception** is an object or piece of data that is thrown from one place (the location of the error) to another (a **catch** statement that contains the appropriate code for handling the situation). The context for exception handling is a **try** block, which is followed by one or more **catch** statements. Here is an arrangement that might be used to handle exceptions.

```
try {
    // to do something
}
catch ( ExceptionType1 & refName ) {
    // handle one ExceptionType
}
...
catch ( ExceptionTypeN & optionalParam ) {
    // statements for processing one of the possible thrown exceptions.
}
```

21.4.2 Exception Types

The keyword `throw` can be followed by an expression of *any type*. In particular, this includes

- Standard exceptions
- Built-in types
- Custom classes

The exception-handling mechanism can transmit information about a run-time problem. Even though it is perfectly legal to throw a piece of data of a simple type (e.g., `int`), this is not regarded as good practice. Instead, it is best to work with `std::exception` objects of classes with descriptive names and a consistent interface that can be used by client code to increase the informational value of `catch` statements.

Example 21.2 contains some custom exception types.

EXAMPLE 21.2 `src/exceptions/example/exceptions.h`

```
[ . . . . ]
/* a custom exception with no data */
class SizeMatchError {
public:
    SizeMatchError() {}
```

```

    QString what() {
        return "Size mismatch for Vector addition!";
    }
};

class BadSizeError { /* a custom exception with data */
public:
    BadSizeError(int sz): m_Sz(sz) {}
    QString what() const {
        return QString("Invalid size for Vector: %1").arg(m_Sz);
    }
    int m_Sz;
};

#include <stdexcept>
using std::exception;
class RangeError : public exception { /* a custom exception
extending from std::exception */
public:
    RangeError() {}
    const char* what() const throw() { ❶
        return "Subscript out of range!";
    }
};
[ . . . ]

```

❶ Matches base class virtual signature.

21.4.3 throwing Things Around

Generally, `throw` statements occur inside function definitions and transmit information to the client code about an exceptional circumstance. The expression inside the `throw` is *copied* into a temporary buffer, making it possible to throw temporary stack objects. A `throw` statement resembles a function call, but really it is more like an `express-return`. This is why.

- A `throw` statement always *returns* information to a position we were before—further back on the stack.
- There is no way to go “back” to the location of the `throw`, because `throw` is going “back” already.
- The stack is *unwound*, meaning that all stack objects are cleaned up, until we reach the stack frame corresponding to a `try/catch` block with a compatible *catch* parameter.
- If a matching `catch` statement is found, its code is executed.
- If no matching `catch` is found, the **default handler** (`terminate()` or `abort()`) is called, which results in the program terminating.

throwing from Constructors and Destructors

For a variety of reasons, exceptions do not mix well with constructors and destructors.

1. Throwing from a destructor is a *very bad idea*, especially if one of those objects is being cleaned up as a result of another `throw`.
2. Throwing an exception from a constructor means that the object's destructor will never be called.

To avoid this, these functions should be **exception-safe**, meaning that *they* catch and handle any possible exceptions that might be thrown.

throw() in a Function Signature

The ANSI/ISO standard permits member function declarations in class definitions to specify which exceptions might be thrown when the function is called. This declaration syntax informs the authors of client code so that they can place `try` blocks and `catch` statements appropriately. A `throw()` expression in a function declaration is part of that function's signature.

The template-based `Vector` class defined in Example 21.3 throws a variety of different kinds of exceptions and demonstrates this feature.

EXAMPLE 21.3 `src/exceptions/example/vector.h`

```
[ . . . . ]

#include "exceptions.h"
using std::bad_alloc;

template <class T> class Vector {
public:
    typedef T* iterator;
    explicit Vector(int n = 100) throw(BadSizeError, bad_alloc);
    Vector(const Vector & v) throw(bad_alloc);
    Vector(const T* a, int n) throw(BadSizeError, bad_alloc);
    ~Vector();
    void display() const;
    iterator begin() const {
        return m_P;
    }
    iterator end() const {
        return m_P + m_Size;
    }
    T& operator[](int i) throw(RangeError);
    Vector& operator=(const Vector& v) throw(bad_alloc);
};
```

```

    Vector operator+(const Vector& v) const throw(SizeMatchError);
private:
    int m_Size;
    T* m_P;
    void copy(const T* a, int n) throw(BadSizeError, bad_alloc);
};

```

The conditions for each `throw` are specified in the member function definitions, shown in Example 21.4. Notice we have function definitions in a header file; this is because they are template functions (see Section 10.1).

EXAMPLE 21.4 `src/exceptions/example/vector.h`

```

[ . . . . ]

template <class T> Vector<T>::
Vector(int n) throw(BadSizeError, bad_alloc) : m_Size(n) {
    if(n <= 0)
        throw BadSizeError(n);
    m_P = new T[m_Size]; ❶
}
[ . . . . ]

template <class T> T& Vector<T>::
operator[](int i) throw(RangeError) {
    if(i >= 0 && i < m_Size )
        return (m_P[i]);
    else
        throw RangeError();
}
[ . . . . ]

template <class T> Vector<T> Vector<T>::
operator+(const Vector& v) const throw(SizeMatchError) {
    if(m_Size != v.m_Size) {
        throw SizeMatchError();
    } else {
        Vector sum(m_Size);
        for(int i = 0; i < m_Size; ++i)
            sum.m_P[i] = m_P[i] + v.m_P[i];
        return sum;
    }
}

```

❶ `new` will throw `bad_alloc` if it fails.

21.4.4 try and catch

Exceptions are raised when certain kinds of operations fail during the execution of a function. If an exception is thrown from within a `try` block (perhaps deeply nested within the block), it can be handled by a `catch` statement with a parameter that is compatible with the exception type.

The syntax of a `try` block has the following form:

```
try compoundStatement handlerList
```

The order in which handlers are defined determines the order that they will be tested against the type of the thrown expression. It is an error to list handlers in an order that prevents any of them from being called.⁴

The `throw` expression matches the `catch` parameter type if it is assignment-compatible with that type.

The syntax of a handler has the following form:

```
catch ( formalArgument ) compoundStatement
```

A `catch` statement looks like the definition of a function that has one parameter but no return type. It is a good idea to declare the *formalArgument* as a reference, to avoid making an unnecessary copy.

If a thrown exception is not caught by an appropriate handler, the default handler will abort the program.

EXAMPLE 21.5 src/exceptions/catch.cpp

```
#include <iostream>
#include <stdlib.h>

using namespace std;

void foo() {
    int i, j;
    i = 14;
    j = 15;
    throw i;
}

void call_foo() {
    int k;
    k = 12;
    foo();
    throw ("This is from call_foo");
}
```

⁴ An example would be having a `catch(QObject&)` before a `catch(QWidget&)`. Since only one `catch` gets executed and the `QObject` is more general than the `QWidget`, it makes no sense to have the `catch(QWidget&)` unless it appears before the `catch(QObject&)`.

```

void call_foo2() {
    double x = 1.3;
    unsigned m = 1234;
    throw (x);
    throw m;
}

int main() {
    try {
        call_foo();
        call_foo2();
    }
    catch(const char* message) { ❶
        cerr << message << endl;
        exit(1);
    }
    catch(int &n) {
        cout << "caught int " << n << endl;
    }
    catch(double& d) { ❷
        cout << "caught a double:" << d << endl;
    }
    catch( ... ) { ❸
        cerr << "ellipsis catch" << endl;
        abort();
    }
}

```

Output:

```

# with the first throw commented out
src/generic> g++ catch.cpp
src/generic> ./a.out
This is from call_foo
src/generic>

# with the first two throws commented out
src/generic> g++ catch.cpp
src/generic> ./a.out
caught a double
src/generic>

# with the first three throws commented out
src/generic> g++ catch.cpp
src/generic> ./a.out
ellipsis catch
Aborted
src/generic>

# with all the throws enabled
src/generic> g++ catch.cpp
src/generic> ./a.out
caught int 14
src/generic>

```

continued

- ❶ Is `const` necessary here?
- ❷ abstract parameter
- ❸ default action to be taken

Example 21.5 demonstrates a few interesting possibilities for handlers. The formal parameter of `catch()` can be abstract (i.e., it can have type information without a variable name). The final `catch(...)` can use an ellipsis and matches any exception type.

The system calls clean-up functions, including destructors for stack objects and for objects local to the `try` block. When the handler has completed execution, if the program has not been terminated, then execution will resume at the first statement following the `try` block.

Example 21.6 shows some client code for the `Vector` template class and exception classes that we defined in the preceding sections. Depending on how much system memory you have on your computer, you may need to adjust the initial values of `BIGSIZE` and `WASTERS` to get this program to run properly. The output is included after the code.

EXAMPLE 21.6 `src/exceptions/example/exceptions.cpp`

```
#include "vector.h"
#include <qstd.h>
using namespace qstd;

void g (int m) {
    static int counter(0);
    static const int BIGSIZE(50000000), WASTERS(6);
    ++counter;
    try {
        Vector<int> a(m), b(m), c(m);
        cerr << "in try block, doing vector calculations. m= "
              << m << endl;
        for (int i = 0; i < m; ++i) {
            a[i] = i;
            b[i] = 2 * i + 1;
        }
        c = a + b;
        c.display();
        if (counter == 2)
            int value = c[m];
        if(counter ==3) {
            Vector<int> d(2*m);
            for(int i = 0; i < 2*m; ++i)
                d[i] = i * 3;
            c = a + d;
        }
    }
```



```

        if(counter == 4) {
            for(int i = 0; i < WASTERS; ++i) ❸
                double* ptr = new double(BIGSIZE);
            Vector<int> d(100 * BIGSIZE);
            Vector<int> e(100 * BIGSIZE); ❹
            for(int i = 0; i < BIGSIZE; ++i)
                d[i] = 3 * i;
        }
    }
    catch(BadSizeError& bse) { ❺
        cerr << bse.what() << endl;
    }
    catch(RangeError& rer) {
        cerr << rer.what() << endl;
    }
    catch(SizeMatchError& sme) {
        cerr << sme.what() << endl;
    }
    catch(...) {
        cerr << "Unhandled error! Aborting..." << endl;
        abort();
    }
    cerr << "This is what happens after the try block." << endl;
}

int main() {
    g(-5); ❻
    g(5);
    g(7);
    g(9);
}

```

Output:

```

src/exceptions/example> ./example
Invalid size for Vector: -5
This is what happens after the try block.
in try block, doing vector calculations. m= 5
<1, 4, 7, 10, 13>
Subscript out of range!
This is what happens after the try block.
in try block, doing vector calculations. m= 7
<1, 4, 7, 10, 13, 16, 19>
Size mismatch for Vector addition!
This is what happens after the try block.
in try block, doing vector calculations. m= 9
<1, 4, 7, 10, 13, 16, 19, 22, 25>
Unhandled error! Aborting...
Aborted
src/exceptions/example>

```

continued

- ❶ Expect `RangeError` to be thrown.
- ❷ Expect `SizeMatchError` to be thrown.
- ❸ Use up most of the available memory.
- ❹ Expect `bad_alloc` to be thrown.
- ❺ Always catch exception objects by reference.
- ❻ Expect `BadSizeError` to be thrown.

Because we did not have a handler for the `bad_alloc` exception, the default handler was called.

EXERCISE: TRY/CATCH

Do some experiments with the code in Example 21.6. For example:

1. What happens if we omit the default handler?
2. What happens if we omit one of the exception types from the `throw()` list in the member function header?

21.4.5 More about throw

Syntactically, `throw` can be used in three ways:

1. `throw expression`. This form raises an exception. The innermost `try` block in which an exception is raised is used to select the `catch` statement that handles the exception.
2. `throw`. This form, with no argument, is used inside a `catch` to rethrow the current exception. It is typically used when you want to propagate the exception to the next outer level.
3. `returnType functionName(arglist) throw (exceptionType list)`. An exception specification is part of the function signature. A `throw` following a function prototype indicates that the exception could be thrown from inside the function body and, therefore, should be handled by client code.

This construct is somewhat controversial. The C++ developer community is split about when it is a good idea to use exception specifications.

In Example 21.7, we make use of the fact that C++ allows composite objects to be thrown. We define a class specifically so that we can throw an object of that type if necessary. We use the quadratic formula to compute one of the roots of a quadratic equation, and we must be careful not to allow a negative value to be passed to the square root function.

EXAMPLE 21.7 `src/exceptions/throw0/throw0.cpp`

```
[....]

class NegArg {
public:
    NegArg(double d) : m_Val(d), m_Msg("Negative value") {}
    QString getMsg() {
        return QString("%1: %2").arg(m_Msg).arg(m_Val);
    }
private:
    double m_Val;
    QString m_Msg;
} ;

double discr(double a, double b, double c) {
    double d = b * b - 4 * a * c;
    if(d < 0)
        throw NegArg(d);
    return d;
}

double quadratic_root1(double a, double b, double c) {
    return (-b + sqrt(discr(a,b,c))/(2 * a));
}

int main() {
    try {
        cout << quadratic_root1(1, 3, 1) << endl;
        cout << quadratic_root1(1, 1, 1) << endl;
    }
    catch(NegArg& narg) {
        cout << "Attempt to take square root of "
             << narg.getMsg() << endl;
    }
    cout << "Just below the try block." << endl;
}
```

Output:

```
-1.88197
Attempt to take square root of Negative value: -3
Just below the try block.
```

The `NegArg` object thrown by the `discr()` function persists until the handler with the appropriate signature, `catch(NegArg)`, exits. The `NegArg` object is available for use inside the handler—in this case to display some information about the problem. In this example, the throw prevented the program from attempting to compute the square root of a negative number.

When a nested function throws an exception, the process stack is “unwound” until an exception handler with the right signature is found.

21.4.6 Rethrown Exceptions

Using `throw` without an expression rethrows a caught exception. The `catch` that rethrows the exception presumably cannot complete the handling of the existing exception, so it passes control to the nearest surrounding `try` block where a suitable handler with the same signature (or ellipsis) is invoked. The exception expression continues to exist until all handling is completed.

If the exception handler does not terminate the execution of the program, execution resumes below the outermost `try` block that last handled the rethrown expression. See Example 21.8.

EXAMPLE 21.8 `src/exceptions/throw2/throw2.cpp`

```
#include <iostream>

void foo() {
    int i, j;
    i = 14;
    j = 15;
    throw i;
}

void call_foo() {
    int k;
    k = 12;
    foo();
}

int main() {
    using namespace std;
    try {
        cout << "In the outer try block" << endl;
        try {
            call_foo(); ❶
        }
        catch(int n) {
            cout << "I can't handle this exception!" << endl;
            throw;
        }
    }
    catch(float z) {
        cout << "Wrong catch!" << endl;
    }
    catch(char s) {
        cout << "This is also wrong!" << endl;
    }
}
```

```

    catch(int n) {
        cout << "\ncaught it " << n << endl;
    }
    cout << "Just below the outermost try block." << endl;
}

```

Output:

In the outer try block
I can't handle this exception!

caught it 14
Just below the outermost try block.

❶ foo exited with i and j destroyed.



Remember that we do not recommend throwing basic types such as int, float, and char. A single number or character conveys little information and does not explain itself to someone reading the code. We violate that rule in some of our examples only to keep them as simple and brief as possible.

21.4.7 Exception Expressions

As we just saw, it is often a good idea to package exception information as an object of a class. The thrown expression can then provide information that the handler can use when it executes. For example, such a class could have several constructors. The `throw` can supply appropriate arguments for the particular constructor that fits the exception.

```

class VectError {
private:
    int m_Ub, m_Index, m_Size;
public:
    VectError(Error er, int ix, int ub); // subscript out of bounds
    VectError(Error er, int sz);       // out of memory
    enum Error { BOUNDS, HEAP, OTHER } m_ErrType;
    . . .
};

```

With such a class definition, an exception can be thrown as follows:

```

throw VectError(VectError::BOUNDS, i, ub);
or
throw VectError(VectError::HEAP, size);

```



Notice that this is a temporary object that is being "thrown," but the `throw` unwinds all stack objects until it gets a matching handler. How can this work?

Exception objects are copied into a special location (not the stack) before the stack is unwound.

It is possible to nest `try` blocks. If no matching handler is available in the immediate `try` block, the search continues, stepwise, in each of the surrounding `try` blocks. If no handler can be found that matches, then the default handler is used, for example, `terminate()`.

EXERCISES: EXCEPTIONS

1. It is often a good idea to organize exception types in hierarchies. This is done for the same reasons that we organize any classes in hierarchies.

Assume that `DerivedTypeError` is derived from `BaseTypeError`. How many errors can you find in the following sequence of handlers?

```
catch(void*)           // any char* would match
catch(char*)
catch(BaseTypeError&) // any DerivedTypeError& would match
catch(DerivedTypeError&)
```

2. As we have seen, you can throw standard exceptions, custom classes, or even basic types (but please don't throw those). Example 21.9 demonstrates the use of custom exceptions inside the scope of a named namespace. This namespace contains the main class definitions.

EXAMPLE 21.9 `src/exceptions/registrar/registrar.h`

```
[ . . . . ]
#include "exceptions.h"
#include <QStringList>

namespace Registrar_Namespace {

    class Student {
    public:
        Student(const QString& name);
        long getNumber() const;
        QString getName() const;
        // other members as needed ...
    };
};
```

```

private:
    long m_Number;           ❶
    QString m_Name;
    static long nextNumber(); ❷
};

class Registrar {
public:
    static Registrar& instance();
    void insert(const Student& stu) throw (DupNumberException);
    void insert(const QString& name);
    void remove(const Student& stu) throw (NoStudentException);
    void remove(const long num)      throw (NoNumberException);
    bool isInList(const Student& stu) const;
    bool isInList(const QString& name) const;

    QStringList report(QString name="all");

    // other members as needed
private:
    Registrar() {};
    Registrar(const Registrar&);    ❸
    Registrar& operator=(const Registrar&);
    QList<Student> m_List;
};
}
[ . . . . ]

```

- ❶ student number
- ❷ used by constructor
- ❸ private constructors

In the next fragment, shown in Example 21.10, we added some exceptions to the same namespace from another header file.

EXAMPLE 21.10 `src/exceptions/registrar/exceptions.h`

```

#ifndef EXCEPTIONS_H
#define EXCEPTIONS_H

#include <QString>

namespace Registrar_Namespace {

    class Exception {
    public:
        Exception (const QString& reason);
        virtual ~Exception () { }
        QString what () const;
    private:
        QString m_Reason;
    };
}

```

continued

```

class NoNumberException : public Exception {
public:
    NoNumberException(const QString& reason);
};

class NoStudentException : public Exception {
public:
    NoStudentException(const QString& reason);
};

class DupNumberException : public Exception {
public:
    DupNumberException(const QString& reason);
};
}

#endif          // #ifndef EXCEPTIONS_H

```

The implementation file shown in Example 21.11 is not complete but there is enough code to get you started.

EXAMPLE 21.11 `src/exceptions/registrar/registrar.cpp`

```

/* Selected implementation examples
   This file is not complete! */

#include "registrar.h"

namespace Registrar_Namespace {

long Student::nextNumber() { ❶
    static long number = 1000000;
    return ++number;
}

Registrar& Registrar::instance() { ❷
    static Registrar onlyInstance;
    return onlyInstance;
}

Exception::
Exception(const QString& reason) : m_Reason(reason) {}

QString Exception::what() const {
    return m_Reason;
}

NoNumberException::
NoNumberException(const QString& reason) : Exception(reason) {}

NoStudentException::
NoStudentException(const QString& reason) : Exception(reason) {}

```



```

DupNumberException::
DupNumberException(const QString& reason) : Exception(reason) {}

}

```

- ❶ Without the above using decl, this would be "long Registrar_Namespace::Student::nextNumber."
- ❷ Implementation of Singleton factory method: This is the only way clients can create instances of this Registrar.

Example 21.12 gives some client code to test these classes.

EXAMPLE 21.12 `src/exceptions/registrar/registrarClientCode.cpp`

```

#include "registrar.h"
#include <qstd.h>

int main() {
    using namespace qstd;
    using namespace Registrar_Namespace;
    Registrar& reg = Registrar::instance();
    while(1) {
        try {
            reg.insert("George");
            reg.insert("Peter");
            reg.insert("Mildred");
            Student s("George");
            reg.insert(s);
            reg.remove(1000004);
            reg.remove(1000004);
            reg.remove(s);
            QStringList report = reg.report();
            foreach (QString line, report) {
                cout << line << endl;
            }
        } catch (NoStudentException& nse) {
            cout << nse.what() << endl;
        }
        catch (NoNumberException& nne) {
            cout << nne.what() << endl;
        }
        catch (DupNumberException& dne) {
            cout << dne.what() << endl;
        }
    }
}

```

- a. There are missing function definitions, which need to be defined before this application is built. Define the missing functions.
- b. Complete and test the client code, and ensure that it works under exceptional circumstances.

REVIEW QUESTIONS

1. What is the difference between a compound statement and a simple statement?
2. How can you guarantee that at least one case will be executed for any given `switch` value?
3. What are the advantages and disadvantages of the three iteration structures? For each, discuss the kinds of situations that would lead you to prefer using it rather than the other two.
4. What happens if a thrown exception is not caught?
5. How can an author of client code know what exceptions need to be handled from a function?
6. How can you catch an unforeseen exception?
7. In response to a throw, what happens to the stack?
8. After an exception is handled, how do we return to the location of the `throw`?
9. Why should we always `catch` objects by reference or pointer?
10. What happens when a destructor throws an exception?
11. What happens when a constructor throws an exception?
12. Which types are preferable for throwing, basic types or object types? Explain the advantages of one over the other.


22

CHAPTER 22

Memory Access

Arrays and pointers are low-level building blocks of C programs that provide fast access to hardware memory. This chapter discusses the different ways to organize and access memory.

22.1	Pointer Pathology	504
22.2	Further Pointer Pathology with Heap Memory	506
22.3	Memory Access Summary	509
22.4	Introduction to Arrays	509
22.5	Pointer Arithmetic	510
22.6	Arrays, Functions, and Return Values	511
22.7	Different Kinds of Arrays	513
22.8	Valid Pointer Operations	513
22.9	What Happens If new Fails?	515
22.10	Chapter Summary	519



Direct manipulation of memory entails some serious risks and requires good practices and thorough testing to avoid serious errors. Improper use of pointers and dynamic memory can cause program crashes that result from heap corruption and memory leaks. Heap corruption is especially difficult to debug because it generally leads to segmentation faults that halt the program at a point in the code that may be far from the point at which the heap became corrupted.

Both Qt and Standard Library container classes permit the safe use of dynamic memory without adversely affecting performance.¹ Arrays are used to implement most container classes but are hidden from client code. The safety factors come from the careful design of each container API so that actions that might produce memory problems are not permitted.

Qt offers many containers, ranging from high-level template classes such as the ones we discussed in Section 10.2 to low-level containers such as `QBitArray` and `QByteArray`.

Generally, when writing applications that reuse those containers, it is easy to avoid the use of arrays entirely. When Qt is not available, or when you are writing an interface to C code, you may need to use arrays and pointers and work directly with allocated memory.

22.1 Pointer Pathology

In Section 1.12 we introduced pointers and demonstrated some of the basics of working with them. We now look at two short code examples to demonstrate some of the weird and dangerous things that can happen when pointers are not handled correctly.

EXAMPLE 22.1 `src/pointers/pathology/pathologydecls1.cpp`

```
[ . . . . ]
int main() {
    int a, b, c;      ❶
    int* d, e, f;    ❷
```

¹ In the sequel, whenever we use the term *container*, with no further qualification, we mean Qt or Standard Library container.

```

    int *g, *h;      ③
    int* i, * j;    ④

    return 0;
}

```

- ① As expected, this line creates three ints.
- ② This line creates one pointer to an int and two ints.(!)
- ③ This line creates two pointers to int.
- ④ This line also creates two pointers to int.

Example 22.1 shows a few of the many ways one can declare pointers. A beginner would be forgiven for thinking the second line of `main()` creates three pointers—after all, in line one, similar syntax creates three integers. However, when multiple variables are declared on one line, the `*` type modifier symbol applies only to the variable that immediately follows it, not the type that precedes it. Since whitespace is ignored by the compiler, the location of whitespace can help or confuse the reader.

EXAMPLE 22.2 `src/pointers/pathology/pathologydecls2.cpp`

```

[ . . . . ]
int main() {
    int myint = 5;
    int *ptr1 = &myint;
    cout << "*ptr1 = " << *ptr1 << endl;
    int anotherint = 6;
    //   *ptr1 = &anotherint;  ①

    int *ptr2;                ②
    cout << "*ptr2 = " << *ptr2 << endl;
    *ptr2 = anotherint;      ③

    int yetanotherint = 7;
    int *ptr3;
    ptr3 = &yetanotherint;  ④
    cout << "*ptr3 = " << *ptr3 << endl;
    *ptr1 = *ptr2;          ⑤
    cout << "*ptr1 = " << *ptr1 << endl;

    return 0;
}
[ . . . . ]

```

- ① error—invalid conversion from `int*` to `int`
- ② uninitialized pointer
- ③ unpredictable results
- ④ regular assignment
- ⑤ dangerous assignment!

Example 22.2 is broken up into three sections. Only the first and third sections are equivalent; the second contains a common beginner's mistake.

```
src/pointers/pathology> g++ pathologydecls2.cpp
pathologydecls.cpp: In function 'int main()':
pathologydecls.cpp:17: error: invalid conversion from 'int*' to 'int'
src/pointers/pathology>
```

After commenting out the invalid conversion, we can try again.

```
*ptr1 = 5
*ptr2 = 1256764
*ptr3 = 7
*ptr1 = 6
```

The value of `*ptr2` is unpredictable.

Dereferencing uninitialized pointers for read purposes is bad enough, but then we *wrote* to it. This is a form of **memory corruption**, which can cause problems later in the program's execution. Notice the inconsistent value that `*ptr1` obtained from `*ptr2`.

22.2 Further Pointer Pathology with Heap Memory

The result of applying `delete` to a pointer that holds the address of a valid object in the heap is to change the status of that heap memory from “in use” to “available.” After `delete` has been applied to a pointer, the state of that pointer itself is *undefined*. The pointer may or may not still store the address of that deleted memory, so a second application of `delete` to the same pointer may cause run-time problems—possibly heap corruption.

In general, the compiler cannot detect attempts to apply `delete` repeatedly to the same object, especially if that piece of memory (or a part thereof) has since been reallocated. To help avoid the very undesirable consequences of a repeated `delete`, it is good practice to assign `NULL` to a pointer immediately after it has been deleted.

If `delete` is applied to a null pointer, there is no action and no error.

Applying `delete` to a non-null pointer that was not returned by `new` produces undefined results. In general, the compiler will not be able to determine whether the pointer was or was not returned by `new`, so undefined run-time behavior can result. Bottom line: *It is the programmer's responsibility to use `delete` correctly.*

One of the richest sources of run-time errors is the production of **memory leaks**. A memory leak is produced when a program causes memory to be allocated and then loses track of that memory so that it can neither be accessed nor deleted.

An object that is not properly deleted will occupy memory until the process terminates.

Some programs (e.g., operating systems) stay active for a long time. Suppose such a program contains a frequently executed routine that produces a memory leak each time it is run. The heap will gradually become perforated with blocks of inaccessible, undeleted memory. At some point a routine that needs a substantial amount of contiguous dynamic memory may have its request denied. If the program was not expecting an event like that, it may not be able to continue.

The operators `new` and `delete` give the C++ programmer increased power as well as increased responsibility.

Here is some sample code that illustrates a memory leak. After defining a couple of pointers, memory will look a little like Figure 22.1.

```
int* ip = new int;           // allocate space for an int
int* jp = new int(13);      // allocate and initialize
cout << ip << '\t' << jp << endl;
```

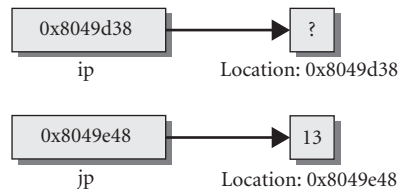


FIGURE 22.1 Initial values of memory

Now we add one more line of code.

```
jp = new int(3);           // reassign the pointer - MEMORY LEAK!!
```

After executing the single line above, our memory looks like Figure 22.2.

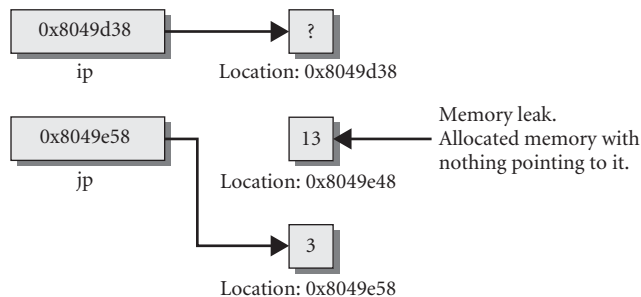


FIGURE 22.2 Memory after leak

EXAMPLE 22.3 `src/pointers/pathology/pathologydemo1.cpp`

```
#include <iostream>
using namespace std;

int main() {
    int* jp = new int(13); ❶
    cout << jp << '\t' << *jp << endl;
    delete jp;
    delete jp;           ❷
    jp = new int(3);     ❸
    cout << jp << '\t' << *jp << endl;
    jp = new int(10);   ❹
    cout << jp << '\t' << *jp << endl;
    int* kp = new int(17);
    cout << kp << '\t' << *kp << endl;
    return 0;
}
```

Output:

```
OOP> g++ pathologydemo1.cpp
OOP> ./a.out
0x8049e08      13
0x8049e08      3
0x8049e08      10
Segmentation fault
OOP>
```

- ❶ allocate and initialize
 - ❷ error: pointer already deleted
 - ❸ Reassign the pointer—MEMORY LEAK!
 - ❹ Reassign the pointer—MEMORY LEAK!
-

In Example 22.3, we deleted the pointer `jp` twice. The second deletion is a serious error but the compiler did not catch it. That error corrupted the heap, made any further memory allocation impossible, and made the behavior of the program beyond that point *undefined*. For example, notice that when we attempted to produce a memory leak by reassigning the pointer `jp`, we did not get any new memory. When we attempted to introduce another pointer variable we got a segmentation fault. This is all undefined behavior and may be different on another platform or with another compiler.

22.3 Memory Access Summary

Here is a list of the most important points that we have raised about memory access.

- The operators `new` and `delete` give the C++ programmer increased power as well as increased responsibility.
- Improper use of pointers and dynamic memory can cause program crashes that result from heap corruption and memory leaks.
- Qt and STL container classes permit the safe use of dynamic memory without adversely affecting performance.
- In a multiple variable declaration, the unary `*` operator applies only to the variable that immediately follows it, not the type that precedes it.
- Dereferencing an uninitialized pointer is a serious error that may not be caught by the compiler.
- After `delete` has been applied to a pointer, the state of that pointer is undefined.
- It is good practice to assign `NULL` to a pointer immediately after it has been deleted.
- Applying `delete` to a non-null pointer that was not returned by `new` produces undefined results.
- The compiler cannot be relied upon to detect the improper use of `delete`, so it is the programmer's responsibility to use `delete` correctly.
- A memory leak is produced when a program causes memory to be allocated and then loses track of that memory so that it can neither be accessed nor deleted.

22.4 Introduction to Arrays

An **array** is a sequence of contiguous memory cells, all of the same size. Each cell is called an array **element**, or entry.

When an array is declared, the size of the array must be made known. This can be done by explicitly or by initialization:

```
int a[10]; // explicitly creates uninitialized cells a[0],
          // a[1], ..., a[9]
int b[] = {1, 3, 5, 7}; // implicitly creates and initializes
                      // b[0], ..., b[3]
```

The array name is an alias for a `const` typed pointer to the first cell of the array. A pointer declaration such as

```
int* ptr;
```

only creates the pointer variable. There is no automatic default initialization of pointer variables. It is an error to attempt to dereference an uninitialized pointer.

Array indices are relative offsets from the base address:

`a[k]` is equivalent to `*(a + k)`

The following bit of code demonstrates an interesting aspect of array indices.

```
#include <iostream.h>
int main(){
    int a[] = {10, 11, 12, 13, 14, 15};
    int* b = a + 1;
    cout << "a[3] = " << a[3] << endl
         << "b[3] = " << b[3] << endl;
}
```

Its output looks like this.

```
a[3] = 13
b[3] = 14
```

Notice that `b` was not declared as an array, but we can use the `[]` operator anyway.

There is a special syntax for defining a dynamic array consisting of a given number of elements of some type.

```
uint n;
ArrayType* pt;
pt = new ArrayType[n];
```

This version of `new` allocates `n` contiguous blocks of memory, each of size `sizeof(ArrayType)` and returns a pointer to the first block. Each element of the newly allocated array is given default initialization. To properly deallocate this array, it is necessary to use the syntax:

```
delete[] pt;
```

Using `delete` without the empty brackets to delete a dynamic array produces undefined results.

22.5 Pointer Arithmetic

The result of applying the operators `+`, `-`, `++`, or `--` to a pointer depends on the type of object pointed to. When an arithmetic operator is applied to a pointer `p` of type `T*`, `p` is assumed to point to an element of an array of objects of type `T`.

- `p+1` points to the next element of the array.
- `p-1` points to the preceding element of the array.
- In general, the address `p+k` is `k*sizeof(T)` bytes larger than the address `p`.

Subtraction of pointers is defined only when both pointers point to elements of the same array. In this case the difference is an `int` equal to the number of array elements between the two elements.

The results of pointer arithmetic are undefined outside the context of an array. It is the responsibility of the programmer to ensure that pointer arithmetic is used appropriately.

EXAMPLE 22.4 `src/arrays/pointerArith.cpp`

```
[ . . . . ]
int main() {
    using namespace std;
    int y[] = {3, 6, 9};
    int x = 23;
    int* px;
    px = &y; ❶
    px = y; ❷
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    return 0;
}
```

❶ The & is redundant here.

❷ y, or any array name, is an "alias" for a pointer to the first element in the array.

If we ran the above example, we might² see something like this:

```
What's next: 6
What's next: 9
What's next: 23
What's next: -1073751080
```

Notice that neither the compiler nor the run-time system reported an error message. C++ happily reads from arbitrary memory addresses and reports them in the type of your choice, giving the C++ developer great power and many opportunities to make great errors.

22.6 Arrays, Functions, and Return Values

As in C, the declared return type of a function cannot be array (e.g., it cannot look like `int []` or `char []` or `Point []`). Returning (addresses to) arrays from functions that are pointer-typed is allowed. However, this is not recommended in the public interface of a class.

An array is a piece of unprotected memory. A class that encapsulates that memory should not have public member functions that return pointers to it. Doing so opens up the possibility for incorrect use of the memory by client code.

² In general, accessing memory beyond the boundary of an array is nonportable, and since that is what we are doing (on purpose), the results will also be nonportable.

A properly designed class completely encapsulates all interactions with any arrays used in the implementation of that class.

Arrays are never passed to functions by value—the array elements will not be copied. If a function is called with an array in its argument list, for example,

```
int a[] = {10, 11, 12, 13, 14, 15};
[ ... ]
f(a[]);
[ ... ]
```

then the array expression is interpreted as a pointer to the first element in the array.

EXAMPLE 22.5 `src/arrays/returningpointers.cpp`

```
#include <assert.h>

int paramSize;

void bar(int *integers) {
    integers[2]=3;
}

int* foo(int arrayparameter[]) {
    using namespace std;
    paramSize = sizeof(arrayparameter);
    bar(arrayparameter);
    return arrayparameter;
}

int main(int argc, char** argv) {
    int intarray2[40] = {9,9,9,9,9,9,9,9,2,1};
    char chararray[20] = "Hello World";
    int intarray1[20];
    int* retval;

    //    intarray1 = foo(intarray2);

    retval = foo(intarray2);
    assert (retval[2] == 3);
    assert (retval[2] = intarray2[2]);
    assert (retval == intarray2);
    int refSize = getSize(intarray2);
    assert(refSize == paramSize);
    return 0;
}
```

- ❶ Change the third element in the incoming array.
- ❷ Pass an array by pointer to a function.
- ❸ Return an array as a pointer from a function.
- ❹ special syntax for initializing char array
- ❺ uninitialized memory
- ❻ uninitialized pointer
- ❼ Error—`intarray1` is like a `// char* const`. It cannot be assigned to.

22.7 Different Kinds of Arrays

Arrays of primitive types, such as `int`, `char`, and `byte`, are used to implement caches. Arrays of objects are supported in the C++ language for backward compatibility with C's arrays of `structs`, but are only used for uniform collections of identical structures, rather than collections of similar polymorphic objects.

If you need random access to the stored items, `QList` (from Qt) or `vector` (from STL) can be used instead of an array. Both are implemented as dynamic arrays under the covers. It is preferable to use those containers in favor of arrays whenever possible, because containers correctly and safely allocate and free memory for you.

22.8 Valid Pointer Operations

Here is a list of the operations that can properly be performed with pointers.

Creation The initial value of a pointer has three possible sources:

- A `const` pointer such as an array name
- An address obtained by using the address-of operator, `&`
- A value obtained by a dynamic memory allocation operator (e.g., `new`)

Assignment

- A pointer can be assigned the address stored by a pointer of the same type or of a derived type.
- A variable of type `void*` can be assigned a pointer of any type without an explicit cast.
- A (non-`void*`) pointer can be assigned the address stored by a pointer of a different (and non-derived) type only with an explicit cast.
- An array name is a `const` pointer and cannot be assigned to.
- A `NULL` pointer (value 0) can always be assigned to any pointer. (Note: Stroustrup recommends that 0 be used instead of `NULL`.)

Arithmetic

- Incrementing and decrementing a pointer: `p++` and `p--`
- Adding or subtracting an integer: `p + k` and `p - k`
- Such expressions are defined only if the resulting pointer value is within the range of the same array. The only exception to this rule is that a pointer is allowed to point to the memory cell that is one position beyond the end of the array as long as no attempt is made to dereference that address.

- For subtraction, two pointers that point to two members of an array can be subtracted, yielding an `int` that represents the number of array elements between the two members.

Comparison

- Pointers to entries of the same array can be compared with `==`, `!=`, `<`, `>`, etc.
- Any pointer can be compared with 0.

Indirection

- If `p` is a pointer of type `T*`, then `*p` is a variable of type `T` and can be used on the left side of an assignment.

Indexing

- A pointer `p` can be used with an array index operator `p[i]` where `i` is an `int`.
- The compiler interprets such an expression as `*(p+i)`.
- Indexing makes sense and is defined only in the context of an array, but the compiler will not prevent its use with non-array pointers where the results are undefined.

The following bit of code in Example 22.6 demonstrates this last point rather clearly.

EXAMPLE 22.6 `src/arrays/pointerIndex.cpp`

```
#include <iostream>
using namespace std;

int main() {
    int x = 23;
    int* px = &x;
    cout << "px[0] = " << px[0] << endl;
    cout << "px[1] = " << px[1] << endl;
    cout << "px[-1] = " << px[-1] << endl;
    return 0;
}
```

Output:

```
src/arays> g++ pointerIndex.cc // compile & run on a Sun station
src/arays> a.out
px[0] = 23
px[1] = 5
px[-1] = -268437516

src/arays> g++ pointerIndex.cc // compile & run on a Linux box
src/arays> ./a.out
```

```
px[0] = 23  
px[1] = -1073743784  
px[-1] = -1073743852  
src/arays>
```

22.9 What Happens If new Fails?



Section 21.4

Every book on C++ has a section on handling new failures. The accepted wisdom for how to handle such failures tends to vary, because the behavior of a C++ program when it runs out of memory is not the same from one platform to another.

We begin our discussion with a caveat. When a C++ program has a memory leak and runs for a long time, eventually there will be no memory available to it. You might think that would cause an exception to be thrown. However most modern operating systems (including *nix and Win32) implement **virtual memory**, which permits the operating system, when its random access memory (RAM) fills up beyond some preset level, to copy the contents of memory that has not been used recently to a special place on the system disk drive. This substitution of relatively slow memory (disk storage) for fast memory (RAM) is generally invisible to the user (except for the performance degradation). If the demands on the system RAM are especially heavy, the OS will use virtual memory to keep satisfying allocation requests until the system starts **thrashing**.³ When this happens, the whole system grinds to a halt until the system administrator can intervene and kill the memory-eating process. At no point will any of the memory allocation failure-handling code be reached in the errant process. It is for this reason that memory allocation errors are handled differently, or not at all, in various applications.

Having said this, the ANSI/ISO standard does specify that the free store operator `new` should throw a `bad_alloc` exception instead of returning `NULL` if it cannot carry out an allocation request. If a thrown `bad_alloc` exception is not caught by a `catch()` block, the default exception handler is called, which could be either `abort()` or `terminate()`.

Example 22.7 demonstrates this feature of C++.

³ When a system is constantly swapping memory back and forth to disk, preventing other I/O from happening, we call that “thrashing.”

EXAMPLE 22.7 `src/newfailure/bad-alloc1.cpp`

```
#include <iostream>
#include <new>
using namespace std;

void memoryEater() {
    int i = 0;
    double* ptr;
    try {
        while(1) {
            ptr = new double[50000000];
            cerr << ++i << '\t' ;
        }
    } catch (bad_alloc& excpt) {
        cerr << "\nException occurred: "
             << excpt.what() << endl;
    }
}

int main() {
    memoryEater(); ❶
    cout << "Done!" << endl;
    return 0;
}
```

Output:

```
src/newfailure> g++ bad-alloc1.cpp
src/newfailure> ./a.out
1         2         3         4         5         6         7
Exception occurred: St9bad_alloc
Done!
src/newfailure>
```

❶ Try to use up the memory.

22.9.1 `set_new_handler()`: Another Approach to new Failures

We can specify what `new` should do when there is not enough memory to satisfy an allocation request. When `new` fails, it first calls the function specified by `set_new_handler()`. If `new_handler` has not been set, a `bad_alloc` object is thrown that can be queried, as shown in Example 22.7, for more information by calling one of its member functions. Example 22.8 shows how to specify our own `new_handler`.

EXAMPLE 22.8 `src/newfailure/setnewhandler.cpp`

```

#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

void memoryEater() {
    int i = 0;
    double* ptr;
    while(1) {
        ptr = new double[50000000];
        cerr << ++i << '\t' ;
    }
}

void out_of_store() {
    cerr << "\noperator new failed: out of store\n";
    exit(1);
}

int main() {
    set_new_handler(out_of_store);
    memoryEater();
    cout << "Done!" << endl;
    return 0;
}

```

Output:

```

src/newfailure> g++ setnewhandler.cpp
src/newfailure> ./a.out
1      2      3      4      5      6      7
operator new failed: out of store
OOP>

```

Note the absence of a `try` block.

EXERCISE: SET_NEW_HANDLER() — ANOTHER APPROACH TO NEW FAILURES

What happens if the last command in the `out_of_store()` function is not `exit()`?

22.9.2 Using `set_new_handler` and `bad_alloc`

Example 22.9 throws a standard exception from the `new_handler`.

EXAMPLE 22.9 `src/newfailure/bad-alloc2.cpp`

```

#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

void memoryEater() {
    int i = 0;
    double* ptr;
    try {
        while(1) {
            ptr = new double[50000000];
            cerr << ++i << '\t' ;
        }
    } catch(bad_alloc& excpt) {
        cerr << "\nException occurred: "
             << excpt.what() << endl;
    }
}

void out_of_store() {
    cerr << "\noperator new failed: out of store\n";
    throw bad_alloc();
}

int main() {
    set_new_handler(out_of_store);
    memoryEater();
    cout << "Done!" << endl;
    return 0;
}

```

Output:

```

src/newfailure> g++ bad-alloc2.cpp
src/newfailure> ./a.out
1         2         3         4         5         6         7
operator new failed: out of store

Exception occurred: St9bad_alloc
Done!
src/newfailure>

```

22.9.3 Checking for null: The Updated Way to Test for new Failures

You may encounter the old null-checking style for detecting failures of `new` in legacy code. That's a sure sign that there are going to be problems with maintenance. Fortunately, there is a simple way to update that old approach.

In Example 22.10, we add the qualifier `(nothrow)` to the allocation statement. As its name suggests, this qualifier suppresses the throwing of `bad_alloc` and allows `new` to return a 0 pointer if it fails.

EXAMPLE 22.10 `src/newfailure/nullchecking.cpp`

```
#include <iostream>
#include <new>
using namespace std;

void memoryEater() {
    int i = 0;
    double* ptr;
    while(1) {
        ptr = new (nothrow) double[50000000];
        if (ptr == 0)
            return;
        cerr << ++i << '\t' ;
    }
}

int main() {
    memoryEater();
    cout << "Done!" << endl;
    return 0;
}
```

Output:

```
src/newfailure> g++ nullchecking.cpp
src/newfailure> ./a.out
1      2      3      4      5      6      7      Done!
src/newfailure>
```

22.10 Chapter Summary

Here is a list of the most important points that we have raised in this chapter.

- An array is a sequence of contiguous memory cells, all of the same size.
- The array name is an alias for a const-typed pointer to the first cell of the array.
- There is no automatic default initialization of pointer variables.
- Array indices are relative offsets from the base address.
- Array subscripts are valid only when used to access members of an array and only within the declared limits of the array.

- The standard does not guarantee that the compiler will catch attempts to use the subscript operator with a pointer that is not an array.
- Arrays are passed to and returned from functions as pointers.
- It is possible to apply the arithmetic operators `+`, `-`, `++`, and `--` to an array pointer, subject to sensible limitations.
- The results of pointer arithmetic are undefined outside the context of an array.
- The standard does not guarantee that the compiler will catch attempts to misuse pointer arithmetic.
- Pointers can acquire values only in the following ways:
 - by initialization when they are first created
 - by assignment after they exist
 - as a result of pointer arithmetic
- A dynamic array of `size` elements of type `T` is allocated using the syntax

```
uint size;
T* pt;
pt = new T[size] ;
```
- Each element of the dynamic array is given default initialization when the array is allocated.
- To deallocate such a dynamic array it is necessary to use the syntax

```
delete[] pt;
```
- The ANSI/ISO standard requires the free store operator `new` to throw a `bad_alloc` exception instead of returning `NULL` if it cannot carry out an allocation request.
- The qualified operator `new (nothrow)` will return `0` if it cannot carry out an allocation request.
- Dynamic arrays should be carefully encapsulated in classes that are designed with proper destructors, copy constructors, and copy assignment operators.

REVIEW QUESTIONS

1. What is defined in the following statement?

```
int* p, q;
```
2. What is a memory leak? How does it happen?
3. How do the +, -, ++, and -- operators work differently on pointers versus regular numbers?
4. What happens if delete is applied to a pointer that has just been deleted?
5. When an array is passed to a function as a parameter, what is being copied onto the stack?
6. What is dynamic memory? How do you obtain it in C++?
7. What happens when a program runs out of memory? How can a program recover from such a situation?
8. Under what situations is it appropriate to check whether a value from new is null?

23

CHAPTER 23

Inheritance in Detail

This chapter formalizes and details some of the concepts introduced earlier in Chapter 6. We explain how constructors, destructors, and copy assignment operators are generated and used by derived classes. We discuss how the keywords `public`, `private`, and `protected` can be used for base classes as well as class members. We also provide examples of multiple inheritance.

23.1 Virtual Pointers and Virtual Tables . . .	524
23.2 Polymorphism and virtual	
Destructors	526
23.3 Multiple Inheritance	528
23.4 public, protected, and private	
Derivation	536

23.1 Virtual Pointers and Virtual Tables

Each class that contains methods (virtual functions) has a virtual jump table, or **vtable**, which is generated as part of the “lightweight” C++ execution environment. The vtable can be implemented in a number of ways, but the simplest implementation (which is often the fastest and most lightweight) contains a list of pointers to all methods of that class. Depending on the optimization policies, it may contain additional information to aid in debugging. The compiler substitutes function names with indirect (relative to the vtable list) references to method calls.

With this in mind, we can define **polymorphic type** explicitly as a class that contains one or more methods and, thus, requires the use of a vtable. Each instance of a polymorphic type has a `typeid`, which can be quite naturally implemented as the address of the vtable for the class.

vtable instead of switch

To implement indirect method calling through vttables, the compiler generates a jump table, which is similar to a `switch` statement, for each polymorphic class. Programmers can often exploit vttables instead of writing their own `switch` statements or large compound conditionals. This is implicit in a number of design patterns such as the Command, Visitor, Interpreter, and Strategy patterns.

A vtable cannot be built for a class unless the method definitions for all overrides are fully defined and findable by the linker.

The `typeid` of an object is set *after* the object’s constructor has executed. If there are base classes, the `typeid` for an object may be set multiple times, after each base class initialization. We will use the classes defined in Example 23.1 to demonstrate that calling a virtual function from a constructor or destructor can have unexpected consequences.

EXAMPLE 23.1 `src/derivation/typeid/vtable.h`

```
[ . . . . ]
class Base {
protected:
    int m_X, m_Y;
public:
    Base();
    virtual ~Base();
    virtual void virtualFun() const;
};

class Derived : public Base {
    int m_Z;
public:
    Derived();
    ~Derived();
    void virtualFun() const ;
};
[ . . . . ]
```

Example 23.2 shows what happens when a virtual function is called from a base class constructor or destructor.

EXAMPLE 23.2 `src/derivation/typeid/vtable.cpp`

```
#include <QString>
#include <qstd.h>
using namespace qstd;
#include "vtable.h"

Base::Base() {
    m_X = 4;
    m_Y = 12;
    cout << " Base::Base: " ;
    virtualFun();
}

Derived::Derived() {
    m_X = 5;
    m_Y = 13;
    m_Z = 22;
}

void Base::virtualFun() const {
    QString val=QString("[%1,%2]").arg(m_X).arg(m_Y);
    cout << " VF: the opposite of Acid: " << val << endl;
}

void Derived::virtualFun() const {
    QString val=QString("[%1,%2,%3]")
```

continued

```

        .arg(m_X).arg(m_Y).arg(m_Z);
        cout << " VF: add some treble: " << val << endl;
    }

Base::~Base() {
    cout << " ~Base() " << endl;
    virtualFun();
}

Derived::~Derived() {
    cout << " ~Derived() " << endl;
}

int main() {
    Base b;
    Derived d;
}

```

In the output that follows, we see that the derived VF: does not get called from `Base::Base()`, because the base class initializer is inside an object that is *not yet* a `Derived` instance.

```

Base::Base: VF: the opposite of Acid: [4,12]
Base::Base: VF: the opposite of Acid: [4,12]
~Derived()
~Base()
VF: the opposite of Acid: [5,13]
~Base()
VF: the opposite of Acid: [4,12]

```

Calling `virtual` methods from destructors is also not recommended. In the preceding output, we can see that the base `virtualFun` is always called from the base class constructors or destructor. Dynamic binding does not happen inside constructors or destructors.

23.2 Polymorphism and virtual Destructors

When operating on classes in inheritance hierarchies, we often maintain containers of base class pointers that hold addresses of derived objects.

Example 23.3 defines a `Bank` class that has a container of various kinds of `Account`.

EXAMPLE 23.3 `src/derivation/assigcopy/bank.h`

```

#ifndef BANK_H
#define BANK_H
#include <QList>
class Account;

class Bank {
public:
    Bank& operator<< (Account* acct); ❶
    ~Bank();
private:
    QList<Account*> m_Accounts;
};
#endif

```

❶ This is how we add object ptrs into m_Accounts.

Bank is able to perform uniform operations on its collected `Account`s by calling virtual methods on each one.

EXAMPLE 23.4 `src/derivation/assigcopy/bank.cpp`

```

[ . . . . ]

#include "bank.h"
#include "account.h"

Bank::~~Bank() {
    foreach (Account* acct, m_Accounts) {
        delete acct;
    }
    m_Accounts.clear();
}

```

In Example 23.4, `delete acct` causes an indirect call to the destructor of `Account`, as well as the subsequent release of allocated memory. However, while every address in the list is an `Account`, some (perhaps all) might point to derived-class objects and therefore require derived-class destructor calls.

If the destructor is `virtual`, the compiler allows run-time binding on the destructor call, instead of simply calling `Account : ~Account()` on each one.

EXAMPLE 23.5 `src/derivation/assigcopy/bank.cpp`

```
[ . . . . ]

Bank& Bank::operator<< (Account* acct) {
    m_Accounts << acct;
    return *this;
}

int main(int argc, char* argv[]) {
    Bank b;
    Account* a1 = new Account(1, 423, "Gene Kelly");
    JointAccount *a2 = new JointAccount(2, 1541, "Fred Astaire",
    "Ginger Rodgers");
    b << a1;
    b << a2;
} ❶
```

❶ At this point, the bank and all the accounts are destroyed.

Without declaring `~Account()` to be `virtual` in the base class, we would get an incorrect result from running Example 23.5.¹

```
Closing Acct - sending e-mail to primary actholder:Gene Kelly
Closing Acct - sending e-mail to primary actholder:Fred Astaire
```

By making the destructor `virtual`, both types of `Account` will get destroyed properly and, in this example, both account holders of a joint account will get proper e-mail notifications when the `Bank` is destroyed.

```
Closing Acct - sending e-mail to primary actholder:Gene Kelly
Closing Joint Acct - sending e-mail to joint actholder:Ginger
Rodgers
Closing Acct - sending e-mail to primary actholder:Fred Astaire
```



If you declare one or more `virtual` methods in a class, you should define a `virtual` destructor for that class, even if it has an empty body.

23.3 Multiple Inheritance

Multiple inheritance is a form of inheritance in which a class inherits the structure and behavior of more than one base class.

¹ Compilers report a missing `virtual` in the destructor as a warning, and the behavior is undefined, so you may not see the same thing on your system.

Common uses of multiple inheritance:

- For crossing the functionality of very different classes with little overlap, such as in Figure 23.1.
- For implementing a common “pure interface” (class with only pure virtual functions) in a variety of different ways.²

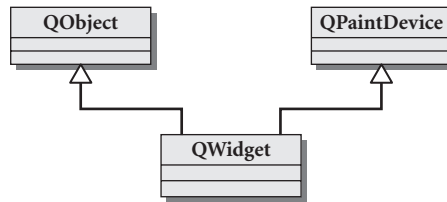


FIGURE 23.1 QWidget’s inheritance

Multiple inheritance hierarchies are more complex and are harder to design, implement, and understand than single inheritance hierarchies. They can be used to solve some difficult design problems, but should not be used if a simpler approach (such as aggregation) is feasible. As with single inheritance, multiple inheritance defines a static relationship among classes. It cannot be changed at runtime.

23.3.1 Multiple Inheritance Syntax

The example in this section demonstrates multiple inheritance syntax and usage.

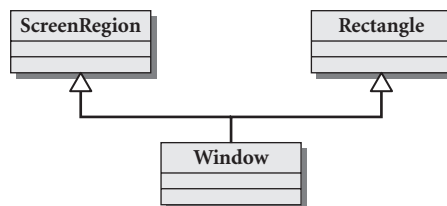


FIGURE 23.2 Window and ScreenRegion

The two base classes shown in Figure 23.2, `Rectangle` and `ScreenRegion`, each have particular roles to play on the screen. One class is concerned with shape and location, while the other is concerned with color and visibility characteristics. A `Window` must be a `Rectangle` *and* a `ScreenRegion`. They are defined in Example 23.6.

² As we see in Section 23.3.2.

EXAMPLE 23.6 `src/multinheritance/window.h`

```

#include "color.h"
#include "point.h"

class Rectangle {
public:
    Rectangle( Const Point& ul, int length, int width);
    Rectangle( const Rectangle& r) ;
    void move (const Point &newpoint);
private:
    Point m_UpperLeft;
    int m_Length, m_Width;
};

class ScreenRegion {
public:
    ScreenRegion( Color c=White);
    ScreenRegion (const ScreenRegion& sr);
    virtual color Fill( Color newColor) ;
    void show();
    void hide();
private:
    Color m_Color;
    // other members...
};

class Window: public Rectangle, public ScreenRegion {
public:
    Window( const Point& ul, int len, int wid, Color c)
        : Rectangle(ul, len, wid), ScreenRegion(c) {} ❶

    Window( const Rectangle& rect, const ScreenRegion& sr)
        : Rectangle(rect), ScreenRegion(sr) {} ❷

    // Other useful member functions ...
};

```

❶ Use base class ctors.

❷ Use base class copy ctors.

There are some syntax items in the *classHead* of the derived class that deserve some attention.

- An access specifier, e.g., `public` or `protected`, must appear before each base class name if the derivation is not `private`.
 - Default derivation is `private`.
 - It is possible to have a mixture of `public`, `protected`, and `private` derivations.

- The comma (,) character separates the base classes.
- The order of base class initialization is the order in which the base classes are listed in the *classHead*.

Client code to put a `Window` on the screen is shown in Example 23.7.

EXAMPLE 23.7 `src/multinheritance/window.cpp`

```
#include "window.h"

int main() {
    Window w(Point(15,99), 50, 100, Color(22));
    w.show();
    w.move (Point(4,6));
    return 0;
}
```

- ❶ calls `ScreenRegion::show()`;
❷ calls `Rectangle::move()`;
-

Member Initialization

Default initialization or assignment proceeds member by member in the order that data members are declared in the class definition: First, base classes; then, derived class members.

23.3.2 Multiple Inheritance with Abstract Interfaces

One situation where it is appropriate to use multiple inheritance is when more than one abstract interface is needed. Figure 23.3 shows a class diagram based on the MP3 Data Model assignment in Section 25.1. `FileTagger` inherits the `DataObject/QObject` for its signals and slots, as well as for its `property()` and `setProperty()` functions. `FileTagger` also needs the `Mp3Song` interface that defines all of the fields for which an ID3 tag should have getters/setters. Similarly, `Mp3File` needs both the `DataObject` and the `Mp3Song` interfaces, even though it is not connected to a physical MP3 file.

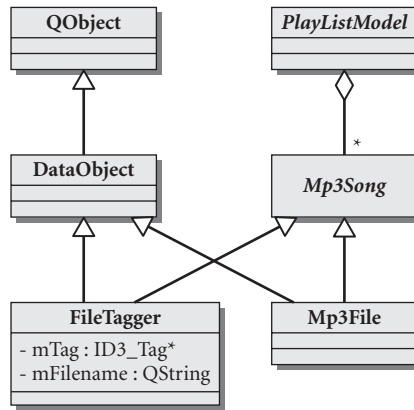


FIGURE 23.3 MP3 data model

Multiple inheritance can help reduce dependencies for client code. For example, a client function can have an `Mp3Song` parameter without needing `id3lib` or `Qt` in order to exploit the `Mp3Song` interface. `Mp3Song`, with only pure virtual functions, enforces the interface on all derived classes. Separating the interface from different implementations makes plugin-frameworks possible.



In Figure 23.3, `QObject` is one of the base classes that is multiply inherited. One restriction `Qt` has is that `QObject` must only be inherited once by each class; further, the `QObject`-derived base must be listed first in the list of base classes. Breaking this rule will lead to strange errors from the code generated by `moc`, the `MetaObject` compiler.

POINT OF DEPARTURE

Multiple inheritance with `QObject` is discussed further in *Qt Quarterly*.³

23.3.3 Resolving Multiple Inheritance Conflicts

Figure 23.4 shows a UML diagram where multiple inheritance is being used incorrectly, for both interface and implementation. To make things even more complicated, we are inheriting from the same base class twice.

³ <http://doc.trolltech.com/qq/qq15-academic.html#multipleinheritance>

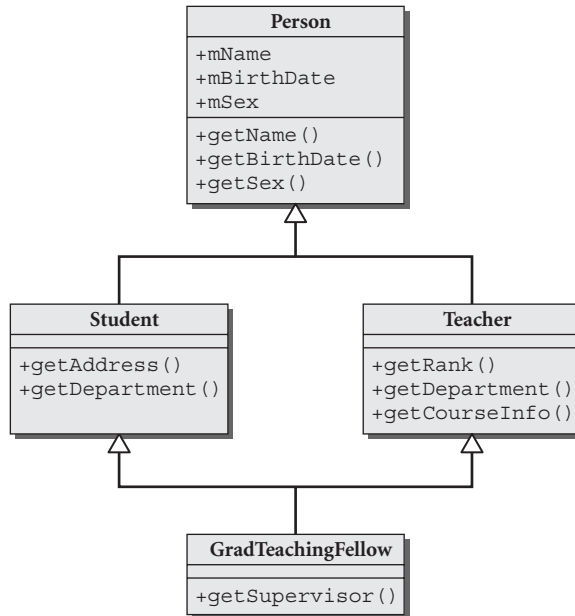


FIGURE 23.4 Person–Student–Teacher

Here, the class `GradTeachingFellow` is derived from two classes: `Student` and `Teacher`.

```

class GradTeachingFellow : public Student,
                          public Teacher {
    // class member functions and data members
};
  
```

Name conflicts and design problems can arise from the improper use of multiple inheritance. In the above example, `getDepartment()` function exists in both `Student` and `Teacher`. The student could be studying in one department and teaching in another, for example.



What happens when you call `getDepartment()` on a `GraduateTeachingFellow`?

```

GraduateTeachingFellow gtf;
Person* pptr = &gttf;
Student * sptr = &gttf;;
Teacher* tptr = &gttf;
gtf.Teacher::getDepartment();
gtf.Student::getDepartment();
sptr->getDepartment()
tptr->getDepartment()
pptr->getDepartment(); // ambiguous - run-time error if virtual
gtf.getDepartment(); // Compiler error - ambiguous function call

```

The problem, of course, is that we have provided no `getDepartment()` function in the `GradTeachingFellow` class. When the compiler looks for a `getDepartment()` function, `Student` and `Teacher` have equal priority.

Inheritance conflicts like these should be avoided because they lead to much design confusion later. However, in this case they can also be resolved with the aid of scope resolution.

23.3.3.1 virtual Inheritance

In Figure 23.4, we inherited more than once from the same base class. There is another problem with that model: redundancy. When we create instances of this multiply inherited class, they might look like Figure 23.5.

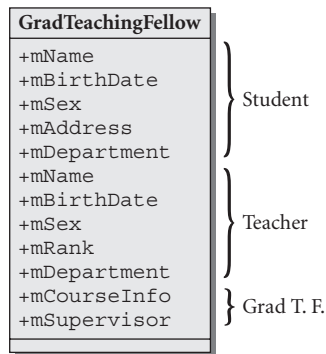


FIGURE 23.5 `GradTeachingFellow`—nonvirtual

`Person` has attributes that we wish to inherit only once. It makes no sense for a `GradTeachingFellow` to have two birthdates and two names. **virtual inheritance** allows us to avoid the redundancy.

The strange problems that can arise when multiple inheritance is used in controversial ways, especially with the added complexities of virtual versus nonvirtual inheritance/functions, seem to have prompted the designers of Java to exclude multiple inheritance from their language. Instead, Java allows the programmer to define an *interface*, which consists only of abstract (pure virtual) functions. A Java class can then use the *implements* clause to implement as many interfaces as it needs.

23.3.3.2 virtual Base Classes

A base class may be declared `virtual`. A virtual base class shares its representation with all other classes that have the same virtual base class.

Add the keyword `virtual` in the `classHead` as shown in Example 23.8, leaving all the other details of the class definitions the same.

EXAMPLE 23.8 `src/multinheritance/people.h`

```
#include "qdatetime.h"

class Person {
public:
    Person(QString name, QDate birthdate)
        QObject(name.ascii()),
        m_Birthdate(birthdate) {}

    Person(const Person& p) : QObject(p),
        m_Birthdate(p.m_Birthdate) {}

private:
    QDate m_Birthdate;
};

class Student : virtual public Person {      ❶
    // other class members
};

class Teacher : virtual public Person {     ❷
    // other class members
}

class GraduateTeachingFellow :
    public Student, public Teacher {       ❸
public:
    GraduateTeachingFellow(const Person& p,
                           const Student& s, const Teacher& t):
        Person(p), Students(s), Teacher(t) {}  ❹
}
```

continued

- ❶ Note keyword `virtual` here.
- ❷ `virtual` inheritance
- ❸ `Virtual` not needed here.
- ❹ It is necessary to initialize all virtual base classes explicitly in multiply-derived classes, to resolve ambiguity about how they should be initialized.

After using `virtual` inheritance, an instance of `GradTeachingFellow` might look like Figure 23.6.

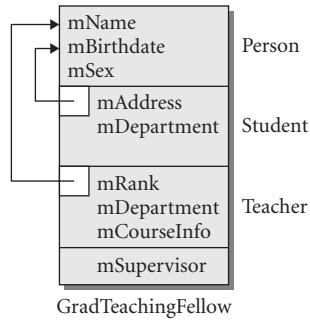


FIGURE 23.6 `GradTeachingFellow`—`virtual`

Each instance of a class that virtually inherits from another has a pointer (or a variable offset) to its virtual base class subobject. The **virtual base class pointer** is invisible to the programmer and, in general, not necessary to change.

With multiple inheritance, each `virtual` base class pointer points to the same object, effectively allowing the base class object to be shared among all of the derived-class “parts.”

For any class with a `virtual` base among its base classes, a member initialization entry for that virtual base must appear in the member initialization for that class. Otherwise, the virtual base gets default initialization.

23.4 `public`, `protected`, and `private` Derivation

Most of the time, you will see classes using `public` derivation. For example,

```
class Square : public Shape {
// ...
};
```

`public` derivation describes an **interface relationship** between two classes. This means that the interface (public part) of the base class merges with the interface

of the derived class. When there is an *is-a* relationship between the derived and base class types, `public` derivation is appropriate.

Much less commonly you will see `protected` or `private` derivation. This is considered an **implementation relationship**, rather than an **is-a** relationship. The base class interface (`public` part) gets merged with the implementation (`private` or `protected`, depending on the kind of derivation) of the derived class. In effect, `private` derivation is like adding an extra object as a `private` data member to your class.

Similarly, `protected` derivation is like adding an object as a `protected` data member to the derived class that shares its `this` pointer.

Example 23.9 is a concrete example of a situation in which `private` derivation might be appropriate. The template class `Stack` is privately derived from `QList`. The rationale for doing this is that a stack is, by definition, a datastructure that limits access to the top item. The class `QStack`, which is publicly derived from `QVector`, has the expected public interface for a stack but it also allows client code unlimited access to the items in the stack because it contains the entire public interface of `QVector`. Our `Stack` class is privately derived from `QList`, so its public interface limits client code access to the handful of stack operations that are consistent with the definition of that data structure.

EXAMPLE 23.9 `src/privatederiv/stack.h`

```
#ifndef _STACK_H_
#define _STACK_H_

#include <QList>

template<class T>
class Stack : private QList<T> {
public:
    bool isEmpty() const {
        return QList<T>::isEmpty();
    }
    T pop() {
        return takeFirst();
    }
    void push(const T& value) {
        prepend(value);
    }
    const T& top() const {
        return first();
    }
}
```

continued

```

    int size() const {
        return QList<T>::size();
    }
    void clear() {
        QList<T>::clear();
    }
};
#endif

```

Example 23.10 shows that an attempt by client code to make use of the `Stack`'s base class (`QList`) interface is not allowed.

EXAMPLE 23.10 `src/privatederiv/stack-test.cpp`

```

#include "stack.h"
#include <QString>
#include <qstd.h>
using namespace qstd;

int main() {
    Stack<QString> strs;
    strs.push("hic");
    strs.push("haec");
    strs.push("hoc");
    //strs.removeAt(2); ❶
    int n = strs.size();
    cout << n << " items in stack" << endl;
    for(int i = 0; i < n; ++i)
        cout << strs.pop() << endl;
}

```

❶ Error—inherited `QList` methods are private.

So, private derivation provides a way to hide the public interface of a base class that was only needed for implementation purposes. What about protected derivation?

Suppose we wish to derive `XStack`, a particular kind of stack, from the `Stack` class. With `Stack` privately derived from `QList`, we will not be able to make use of any `QList` member functions when we implement `XStack`.

If we need to use some of the `QList` functions when we implement `XStack`, then we must use protected derivation when we derive `Stack` from `QList`.

protected derivation makes the public interface of `QList` protected in `Stack`.

Internally, this allows classes derived from `Stack` to make use of the inherited `QList` protected interface.

REVIEW QUESTIONS

1. What is a vtable?
2. What is a polymorphic type?
3. Which kinds of member functions are not inherited? Why?
4. Under what circumstances should we have virtual destructors?
5. What happens when a `virtual` function is called from a base-class constructor?
6. What is `virtual` inheritance? What problems can it be used to solve?
7. Why would one use non-public derivation?

24

CHAPTER 24

Miscellaneous Topics

Variable length argument lists are referenced, but not completely explained earlier in the book, so they are discussed in this chapter. An example of resource sharing is also presented.

24.1 Functions with Variable-Length Argument Lists	542
24.2 Resource Sharing	543

24.1 Functions with Variable-Length Argument Lists

In C and in C++ it is possible to define functions that have parameter lists ending with an ellipsis (...). The ellipsis allows the number of parameters and their types to be specified by the caller. The usual example of such a function is from `<stdio.h>`.

```
int printf(char* formatstr, ...)
```

This flexible mechanism permits calls such as

```
printf("Eschew Obfuscation!\n");  
printf("%d days hath %s\n", 30, "September");
```

To define a function that uses the ellipsis you need to

```
#include <cstdarg>
```

which adds to the `std` namespace a set of macros for accessing the items in the argument list. There must be at least one parameter other than the ellipsis in the parameter list. A variable, usually named `ap` (argument pointer), of type `va_list` is used to traverse the list of unnamed arguments. The macro

```
va_start(ap, p)
```

where `p` is the last named parameter in the list, initializes `ap` so that it points to the first of the unnamed arguments. The macro

```
va_arg(ap, typename)
```

returns the argument that `ap` is pointing to and uses the `typename` to determine (i.e., with `sizeof`) how large a step to take to find the next argument. The macro

```
va_end(ap)
```

must be called after all of the unnamed arguments have been processed. It cleans up the unnamed argument stack and ensures that the program will behave properly after the function has terminated.

Example 24.1 shows how to use these features.

EXAMPLE 24.1 `src/ellipsis/ellipsis.cpp`

```

#include <cstdarg>
#include <iostream>
using namespace std;

double mean(int n ...) {
    va_list ap;
    double sum(0);
    int count(n);
    va_start(ap, n);
    for(int i = 0; i < count; ++i) {
        sum += va_arg(ap, double);
    }
    va_end(ap);
    return sum / count;
}

int main() {
    cout << mean(4, 11.3, 22.5, 33.7, 44.9) << endl;
    cout << mean (5, 13.4, 22.5, 123.45, 421.33, 2525.353) << endl;
}

```

- ❶ First parameter is number of args.
- ❷ Sequentially points to each unnamed arg.
- ❸ ap now points to first unnamed arg.
- ❹ Clean up before returning.

24.2 Resource Sharing

Garbage collection is a process that recovers heap memory that is no longer being referenced. Languages such as LISP, Smalltalk, and Java have built-in garbage collectors that run in the background and track object references. When an object is no longer referenced it is deleted, and the memory that it occupied is made available for use by other objects.

The next examples show a way of building garbage collection into the design of a class by means of **reference counting**. Reference counting is an example of **resource sharing**.

Each object keeps track of its active references. When an object is created, its reference counter is set to 1. Each time the object is newly referenced, the reference counter is incremented. Each time it loses a reference, the reference counter is decremented. When the reference count becomes 0, the shared object can be deallocated.



WHAT ABOUT CHANGES? If the object is about to be changed (e.g., a non-const member function is called) and its reference count is greater than 1, it must be cloned first so that it is no longer shared.

In Example 24.2, we define a homemade string class, `MyString`, that contains a private inner class, `MyStringPrivate`, which is responsible for the creation of dynamic arrays and for maintaining a reference count.

An inner class is simply a class defined inside another class. Inner classes are “private” classes, meant to be used only by the containing class.

EXAMPLE 24.2 `src/mystring/refcount/refcount.h`

```
[ . . . . ]

class MyString {
    class MyStringPrivate {
        friend class MyString; ❶
    public:
        MyStringPrivate() : m_Len(0), m_RefCount(1) {
            m_Chars = new (nothrow) char[1] ;
            m_Chars[0] = 0;
        }
        MyStringPrivate(const char* p) : m_RefCount(1) {
            m_Len = strlen(p);
            m_Chars = new (nothrow) char[m_Len + 1];
            if (m_Chars)
                strncpy(m_Chars, p, m_Len + 1);
            else
                cerr << "Out of memory in MyStringPrivate ctor!"
                    << endl;
        }
        ~MyStringPrivate() {
            delete []m_Chars;
        }
    private:
        int    m_Len, m_RefCount;
        char*  m_Chars;
};

public:
    MyString() : m_Impl(new MyStringPrivate) {}
    MyString(const char* p)
        : m_Impl(new MyStringPrivate(p)) {}
    MyString(const MyString& str);
    ~MyString();
```

```

    void operator=(const MyString& str);
    void display() const ;
    int length() const;
private:
    MyStringPrivate* m_Impl;
};
[ . . . . ]

```

❶ Even though this is an inner class, we need to give friend permissions to the containing class.

nothrow

We used the `nothrow` qualifier for `new` (Section 22.9.3) to avoid having to add exception handling code to the example.

The public class `MyString`, because it manages shared instances of `MyStringPrivate`, is sometimes called a **handler class**. It is responsible for maintaining the correct value of the reference counter, and for deleting the pointer when the counter reaches zero.

In Example 24.3, we have output statements in the definitions of three member functions to show the reference counter as objects are created and destroyed.

EXAMPLE 24.3 `src/mystring/refcount/refcount.cpp`

```

[ . . . . ]
MyString::MyString(const MyString& str) : m_St(str.m_St) {
    m_St -> m_RefCount++;
    cout << m_St->m_S << "::refcount: " << m_St->m_RefCount << endl;
}

MyString::~MyString() {
    cout << m_St->m_S << "::refcount: " << m_St->m_RefCount << endl;
    if (--m_St -> m_RefCount == 0) {
        cout << m_St->m_S << "::memory released" << endl;
        delete m_St;
    }
}

void MyString::operator=(const MyString& str) {
    if (str.m_St != m_St) {
        if (--m_St -> m_RefCount == 0)
            delete m_St;
        m_St = str.m_St; ❶
    }
}

```

continued

```

        ++(m_St->m_RefCount);
    }
}
[ . . . . ]

```

❶ Just copy the address.

The client code shown in Example 24.4 contains a function with a value parameter and a `main()` with an inner block. Inside the block, objects are created, copied, and destroyed.

EXAMPLE 24.4 `src/mystring/refcount/refcount-test.cpp`

```

#include "refcount.h"
void fiddle(MyString lstr1) {
    cout << "inside fiddle()" << endl;
    MyString lstr2(lstr1);
    MyString lstr3;
    lstr3 = lstr2;
}

int main() {
    MyString str1("AABBCCDD");
    {
        cout << "local block begins" << endl;
        MyString str2(str1);
        fiddle(str2);
        cout << "back from fiddle()" << endl;
    }
    cout << "local block ends" << endl;
    str1.display();
    cout << endl;
    return 0;
}

```

The output that follows dispassionately shows the entire saga of birth and death as the process races from opening brace to closing brace.

```
local block begins
AABBCCDD::refcount: 2
AABBCCDD::refcount: 3
inside fiddle()
AABBCCDD::refcount: 4
AABBCCDD::refcount: 5
AABBCCDD::refcount: 4
AABBCCDD::refcount: 3
back from fiddle()
AABBCCDD::refcount: 2
local block ends
AABBCCDD
AABBCCDD::refcount: 1
AABBCCDD::memory released
src/mystring>
```

EXERCISES: RESOURCE SHARING

1. Example 24.3 demonstrates reference counting but does not deal with the question of what to do if a `MyString` object needs to have its value changed when its reference counter is greater than 1. How would you implement the cloning of a `MyString` object when necessary (but *only* when necessary)? Implement your solution and test it.
2. Implement a thread-safe version using `QMutex`.
3. Rewrite `MyString` using `QSharedData` and `QSharedDataPointer`.



P A R T I V

Programming Assignments



Chapter 25. MP3 Jukebox Assignments 551

25

CHAPTER 25

MP3 Jukebox Assignments

In the assignments in this chapter, we will write, in stages, a program that serves as an MP3 playlist generator and database manager. It will generate and play selections of MP3 songs based on what it can find on our file system and it will permit filter-queries based on data stored in ID3v2 (meta) tag information.

25.1 Data Model: Mp3File	553
25.2 Visitor: Generating Playlists.....	555
25.3 Preference: An Enumerated Type	556
25.4 Reusing id3lib	559
25.5 PlayListModel Serialization	560
25.6 Testing Mp3File Related Classes.....	561
25.7 Simple Queries and Filters	561
25.8 Mp3PlayerView.....	563
25.9 Models and Views: PlayList	565
25.10 Source Selector.....	566
25.11 Persistent Settings.....	567
25.12 Edit Form View for FileTagger.....	568
25.13 Database View	569

The features we will implement are inspired by open-source programs such as amaroK and Juk (and commercial programs such as iTunes and MusicMatch Jukebox). These programs all provide similar features and similar styles of user interface.

The code that does the actual file-tagging is taken from an open-source library, `id3lib`¹ version 3.8.3. This is the same library that is used by MusicMatch Jukebox.

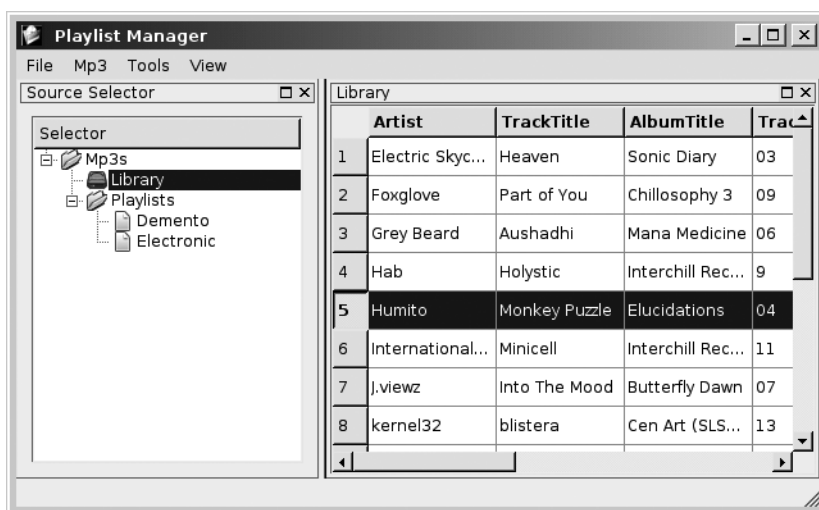


FIGURE 25.1 Example screenshot

The media player shown in Figure 25.1 has three major components.

1. A player view that shows the user what is currently playing and provides some controls for changing volume or position of the song
2. A selector that permits the user to choose (and create new) playlists for manipulating or playback
3. A song list view, for displaying a list of songs in a tabular form

¹ <http://sourceforge.net/projects/id3lib>

Each of these major components has a view for displaying the data and a model for storing the data.

- The model for a `Mp3PlayerView` consists of `Mp3Player` plus a `FileTagger`.
- The model for a source selector is a tree, and you can use either `QTreeWidgetItem` or `QAbstractItemModel` as the base class for this.
- To start with, the model for a song list view will be a simple `Playlist`. Later in the chapter, we will implement another model based on `QSqlRelationalTableModel`.

Figure 25.2 shows a high-level UML diagram of the major components of this project. We can see that the `Controller` class, derived from `QApplication`, owns all other objects.

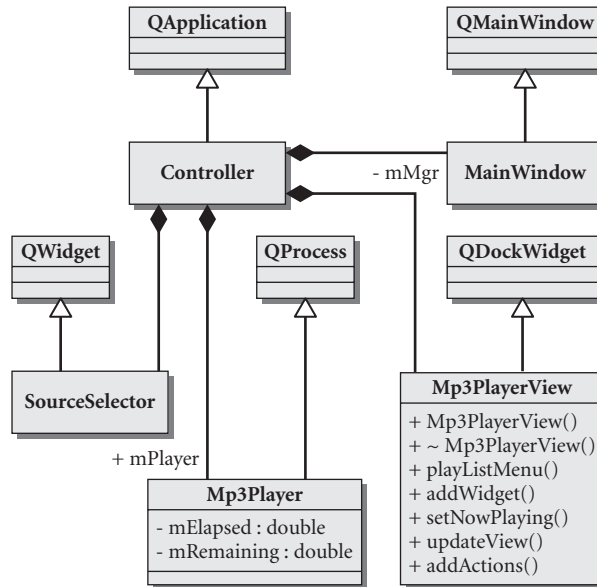


FIGURE 25.2 The Controller and its managed objects

25.1 Data Model: Mp3File



- Abstract base classes (Section 6.3)
- Multiple inheritance (Section 23.3)

Figure 25.3 shows a UML diagram of the data model for our MP3 player application, at the lowest level.

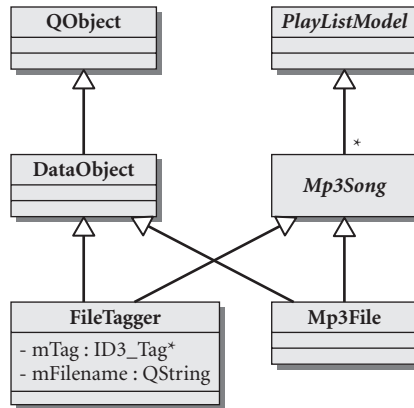


FIGURE 25.3 Initial data model

Mp3Song is an abstract class (which is why it is italicized in the UML diagram). This interface is meant to describe a common set of features for different implementations of the Mp3Song interface.

EXAMPLE 25.1 `../src/libs/filetagger/mp3song.h`

```

#ifndef _MP3SONG_H
#define _MP3SONG_H

#include <qobject.h>
#include <qstring.h>
#include <QStringList>

class Mp3Song {
public:
    static QStringList fields();
    virtual ~Mp3Song();

    virtual QString getGenre() const =0;
    virtual QString getArtist() const =0;
    virtual QString getAlbumTitle() const =0;
    virtual QString getTrackTitle() const =0;
    virtual QString getTrackNumber() const =0;
    virtual int getTrackTime() const =0;
    virtual QString getComment() const =0;
    virtual QString getPreference() const =0 ;
    virtual QString toString() const =0;
    virtual QString getUrl() const =0;
    virtual QString getFilename() const = 0;

```

```

virtual void setPreference(const QString & newPref) = 0;
virtual void setGenre (const QString& newGenre) =0;
virtual void setArtist (const QString& newArtist) =0;
virtual void setTrackNumber (const QString& trackNum) = 0;
virtual void setTrackTitle (const QString& newTitle) = 0;
virtual void setAlbumTitle (const QString& newAlbumTitle) = 0;
virtual void setComment (const QString& newComment) = 0;
virtual void setFilename (const QString& newFilename) =0 ;

};

#endif

```

Note also that there is a `getTrackTime()` function but no `setTrackTime()` method. This is because `m_TrackTime` is a calculated value, a read-only property. Derived classes will override this method and return the actual time of the song.

The Assignment

- Define a class `Mp3File` that implements the `Mp3Song` interface but allows you to store and retrieve the values in data members.
- Define a `PlayListModel`, a collection of `Mp3Songs`. Reuse a `QList<Mp3Song*>` to hold pointers to heap `Mp3File` objects.
- Write the implementations to the classes described in the UML diagram in Figure 25.3. This includes data members, getters and setters, and a `toString()` method for `Mp3File`.

25.2 Visitor: Generating Playlists



- Command line arguments (Section 1.8.1)
- Visitor pattern (Section 8.1)

In the `http://oop.mcs.suffolk.edu/dist` folder, you will find `filetagger.tar.gz`, which contains interfaces `PlayListModel` and `Mp3Song`, as well as other classes you might reuse for future assignments. The assignment is to write a `Playlist` class and a program that generates instances of them by scanning directories for MP3 files. The program should be called `playgen`. Figure 25.4 shows how the interfaces are related to `Playlist`.

playgen (the application) should have a class called `PlayGen`, containing a `PlayList* scan(QString dirname)` that returns a new `PlayList` containing references to each song.

Usage:

```
playgen dirname
```

The program should print out the name of each file that it finds (on a line by itself). But it should do this by first loading the `PlayList` and then displaying the string produced by `PlayList::toString()`.



Classes you can reuse: `QFile` and `QFileInfo`, or `FileVisitor`.

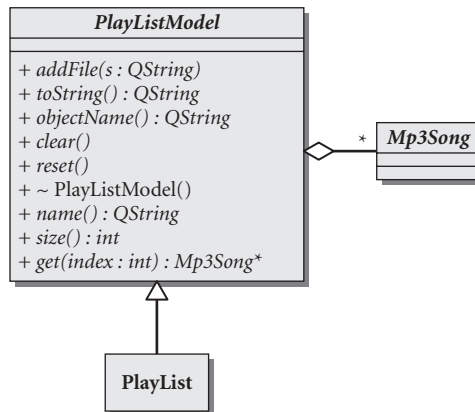


FIGURE 25.4 `PlayList` and `PlayListModel`

25.3 Preference: An Enumerated Type



- Enumerations (Section 19.3)
- Types (Section 19.5)
- Conversions (Section 19.6)

Your assignment is to design, implement, and test a `Preference` class.

Preference is a class that models one field in an ID3 tag. It is intended to permit the user to specify the quality of an MP3 file (e.g., excellent, good, fair, poor). We need a uniform preference system that will enumerate a fixed set of value choices so that comparisons and subrange queries are possible.

Here is the public interface of the class definition to get you started.

```
public:
    Preference(int value=0);
    Preference(QString prefstr);
    /**
     * If possible, set the host value from the given string and
     * @return true. Otherwise return false.
     */
    bool fromString(QString);
    /**
     * @return a list of all the acceptable Preference names,
     *         ordered by value (increasing)
     */
    virtual QStringList getNames() const;
```

The Assignment

- Complete the class definition in `preference.h`.
- Implement all the functions in `preference.cpp` so that your class can pass the test case below.
- Write a `main.cpp` that calls this test case and tests your `Preference` class.
- Generate or write a `qmake` project file that can build the application.
- Verify that `make dist` creates a `dist` target (a tarball).

A test case that shows how the `Preference` class must work is provided in Example 25.2. Your `Preference` class must pass that test.

EXAMPLE 25.2 `../src/libs/filetagger/testpreference.cpp`

```
#include "testpreference.h"
#include "preference.h"
#include "qstd.h"
using namespace qstd;

void TestPreference::test() {

    Preference verygood("Very Good");
    Preference verygood2("Very Good");
    Preference excellent("Excellent");
    Preference fair("Fair");
    Preference good("Good");
```

continued

```

Preference none("None");
Preference poor("Poor");
Preference badtaste("Bad Taste");
Preference undefined("undefined");
ASSERT_EQUALS(verygood, verygood2);
ASSERT_NOTEQUALS(verygood, fair);

ASSERT_EQUALS(undefined, 0);
ASSERT_TRUE(none > undefined);
ASSERT_TRUE(poor < none);
ASSERT_TRUE(badtaste < poor);
ASSERT_TRUE(verygood > good);
ASSERT_TRUE(good > fair);
ASSERT_TRUE(fair > none);
ASSERT_TRUE(fair < verygood);
ASSERT_TRUE(verygood < excellent);

ASSERT_EQUALS(verygood.toString(), "Very Good");

qDebug() << verygood.toString();
Preference q("notsogood");
ASSERT_EQUALS(0, (int)q);
qDebug() << q.getNames().join(", ");

/* Optional - Case Ignore conversions? */

// Does this print "fair is fair" or "fair is 4"?
cout << "Fair is " << fair;

Preference verygoodlc("very good");
ASSERT_EQUALS(verygood, verygoodlc);
}

```

The header file is provided to you in Example 25.3 for completeness.

EXAMPLE 25.3 `../src/libs/filetagger/testpreference.h`

```

#include <testcase.h>
#include <assertequals.h>

class TestPreference : public TestCase {
    TestPreference() : TestCase("TestPreference") {}

protected:
    virtual void test() ;
};

```

When the test case is run, the output should look like this.

```

<testcase>
Very Good
Undefined, Bad Taste, Poor, None, Fair, Good, Very Good, Excellent

```

```
./testpreference.cpp:44: assertequalsfailed:
    expr var="verygood" val="6" ,    expr var="verygoodlc" val="0"
    <testcaseinfo classname="TestCase" name="unnamed" status="passed" />
</testcase>
Fair is Fair
```

25.4 Reusing id3lib

Each MP3 file contains meta-data called “ID3 tags” that can be accessed and manipulated through low-level file operations, or via a higher-level interface. We saw two interfaces for using the library in Section 16.3. For this exercise, we have provided you with classes from three libraries (<http://oop.mcs.suffolk.edu/dist>) and you are to write a program that reuses them all.



- Reusing libraries (Chapter 7)
- Visitor pattern (Section 8.1)
- Command line arguments (Section 6.7.1)
- id3lib (Section 16.3)
- playgen (Section 25.2)

ID3 tags enable you to read and store meta-information about MP3 songs in a special part of the MP3 file itself. To manipulate MP3 files as though they were C++ objects, getting and setting properties that persist in the ID3 tags, we reuse an ID3 library. You will be provided with a `FileTagger` class that depends on `id3lib`, but implements the `Mp3Song` interface.



Check out some of the executables that come with `id3lib`: `id3info`, `id3convert`, `id3tag`, `id3cp`. They all have man pages.

The Assignment

1. Install `id3lib`.
2. Build `filetagger.cpp` with `TestFileTagger.cpp` and run it on a folder with some junk MP3 files that you do not care about. Verify that the test case passes.

3. Enhance the `playgen` program so that its output playlist includes ID3 tag information in the playlist.

```
playgen [directory]
playgen [file1.mp3] [file2.mp3] ...
```

`playgen` generates a playlist of all MP3 files in the directory, or of all MP3 files supplied on the list of command line arguments. It should read the `id3tag` information from each file it visits and display each song in the format shown in Example 25.4.

EXAMPLE 25.4 `filetagger-examplefiles/test-winamp-playlist.m3u`

```
#EXTM3U
#EXTINF:134, Tom Lehrer - Vatican Rag
Tom Lehrer\YearThatWas\14_Vatican Rag_Tom Lehrer.mp3
#EXTINF:131, Tom Lehrer - Folk Song Army
Tom Lehrer\YearThatWas\04_Folk Song Army_Tom Lehrer.mp3
#EXTINF:155, Tom Lehrer - National Brotherhood Week
Tom Lehrer\YearThatWas\01_National Brotherhood Week_Tom Lehrer.mp3
```

It is suggested you use `ArgumentList` to process command line options, and `FileVisitor` to visit each file.

25.5 PlaylistModel Serialization



- Streams and files (Section 1.10)
 - Serializer pattern (Section 10.6)
-

Most MP3 players load/save their playlists from/to a file with an M3U extension. There is simple M3U and extended M3U (EXTM3U). Simple M3U is just a list of filenames separated by newlines. Extended M3U contains extra information following a `#` sign on the preceding line. If you use this format, your program will immediately work with the most popular MP3 players.

Serialization, as we discussed in Section 10.6, is the process of taking an object and expressing its state information in a format that allows it to be sent across a network or saved to a storage device such as a file or a database. **Deserialization** is the process of reconstructing the object from the serialized state information to its original state.

Now that we have a simple data object, we want to be able to serialize and deserialize it. Write two classes, `PlayListReader` and `PlayListWriter` for reading/writing playlist data.

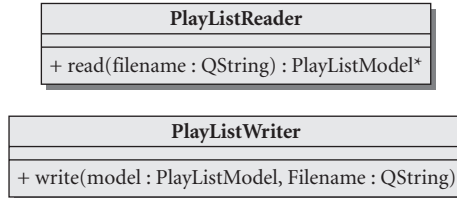


FIGURE 25.5 Serialization classes for `PlayListModel`

25.6 Testing Mp3File Related Classes

- Define a class `TestMp3File` and a member function, `test()`.
- The test case should do the following:
 1. Using the classes from `playgen`, create a `PlayList` from a directory specified on the command line, visiting each file.
 2. Using `PlayListWriter`, write the `PlayList` to a file.
 3. Using `PlayListReader`, read the file into another `PlayList` instance.
 4. Write an `equals()` function for `PlayList` that tests that each property of the `Mp3File` in the collection has the same value as the corresponding one in the original.

25.7 Simple Queries and Filters



- Abstract Factory (Section 16.1)
- Regular expressions (Section 13.2)
- Visitor (Section 8.1)

This exercise is to enhance `playgen`, so that it generates a `PlayList` based on the tag values of visited `Mp3Files`. To achieve this, we need multi-dimensional property constraints.

Two classes from `libdataobjects` can help you solve this problem. `ConstraintGroup` is a class you can reuse for multidimensional Constraints.

Each Constraint is a piece of a query, so a query can be represented by a ConstraintGroup.

```
usage: playgen [options] filespec
[options] are optional.
filespec can be a directory to scan,
        or a list of mp3 files to add to playlist.
Directories are recursed. Additional options can be:
    (filter options)
    -p Good - filter on Preference
    -p 5 - equivalent to preference Good
    -p "Very Good" - Need double quotes around this one
    -g Chill - filter on Chill genre
    -a ".*gabriel.*" - regex filter on artists - need
        doublequotes around regex
    -b albumpattern - filter on album title
    -s songpattern - filter on song title
```

Advanced queries:

If the same switch appears multiple times, OR the values together.

```
-p Excellent -p "Very Good" (either can be true)
```

If different switches appear on the same query AND them together.

```
-p Excellent -g Ambient (both must be true).
```

```
-p "6,7" - Allow preferences 6 and 7 only
```

```
-p "4:9" - Filter on preferences of the subrange 4 to 9
```

```
-p "0,5:" - filter on preferences undefined, or anything
        better than "good"
```

```
-p "1,2,3" - only preferences "badtaste", "poor" and "none"
```

example:

```
playgen ./music/comedy/weirdal
```

make a playlist of all the songs in that directory, without any filtering, and send to standard output.

```
playgen -o "weirdalsbest.m3u" -p "Excellent" ./music/
comedy/weirdal
```

Should go into the weirdal directory and spit out only the "Excellent" tracks, saving them to a file.

```
playgen -g "(Rock|Classical|Dub)" ./music/techno
```

Filter on genres - since you're using regular expressions to match, you have the full regular expression query language here.

In Figure 25.6, the UML diagram shows one possible way of organizing your code. You are free to deviate from this diagram as you design your own solution.

You might think of your program in the following way.

1. Given an `ArgumentList`, create an object that represents the collection of field/value pairs as a set of constraints.
2. Use `FileVisitor` to visit each file in `filespec`.

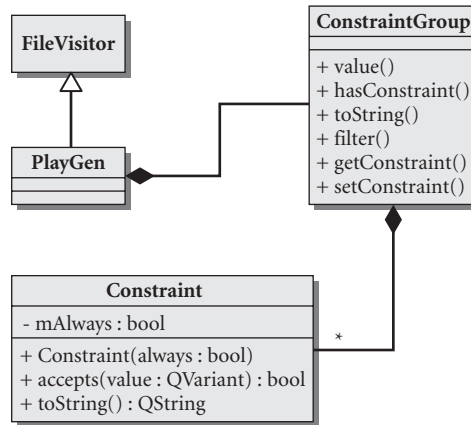


FIGURE 25.6 PlaylistGenerator with constraints

3. Use `FileTagger` to extract the ID3 information of each MP3 file.
4. After you have extracted the ID3 information, check whether the object satisfies the constraints specified.
5. If the object satisfies your constraints, we want to make a copy of the `FileTagger`'s attributes in a new `Mp3File` object and add the `Mp3File` to the `Playlist` returned by `playgen` (instead of adding the actual `FileTagger`, which we will reuse for the next visited file).
6. Return a `Playlist` with the selected songs.
7. Write a factory for `Query` objects, called `QueryFactory`, and a function, called `newQuery`, to handle all instance creation.

```

class QueryFactory {
    Query * QueryFactory::newQuery(QStringList2 args);
};
  
```

25.8 Mp3PlayerView

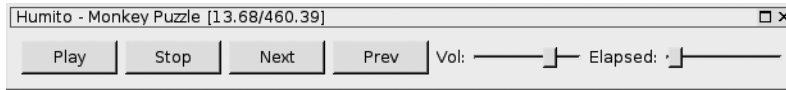


- Widgets (Chapter 11)
- Signals and slots (Section 9.3.3)
- `QProcess` (Section 12.1)

² or `ArgumentList`

The minimal MP3 player view has the following features:

- Buttons: play, stop, next, and previous
- Sliders: volume control and song progress
- Label: to display “now playing” information



An example `Mp3Player` class based on Figure 25.7 is supplied to you. This particular implementation works by creating `QProcess` objects, running command-line programs, parsing the output, and displaying information based on it. It uses `mpg321`³ (for MP3 decoding) and `alsaplayer`⁴ (for volume control). You can reuse this class if you are using Linux.

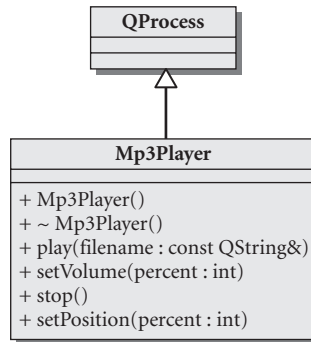


FIGURE 25.7 `Mp3Player`

1. Write another `Mp3Player`-derived class, better suited to your own platform or personal needs. It can be based on <http://www.music.mcgill.ca/~gary/rtaudio/index.html#license> or <http://www.speech.kth.se/snack>, two open-source libraries for playing media files. Alternately, you can write it as a plugin for an already existing media player, such as `xmms` or `Winamp`.
2. Write a front-end to this `Mp3Player` class that allows you to load songs, start/stop, change volume, and see the play progress advance in the slider.
3. Enhance `Mp3Player` and `PlayerView` so that you can change the position of the song playing by manipulating the song progress slider.

³ <http://mpg321.sourceforge.net>

⁴ <http://www.alsaplayer.org>

25.9 Models and Views: PlayList



- Qt 4 models and views (Section 17.3)
- `QActions` and `QMenus` (Section 11.6)

This application has a “central widget” that should display the current playlist. Sometimes it will be a view of a `PlayList`, but other times, it will show us the contents of a `Database`. One approach, shown in Figure 25.8 is to make both classes multiply-inherit from `QAbstractTableModel`, so they can both be viewed in a `QTableView`.

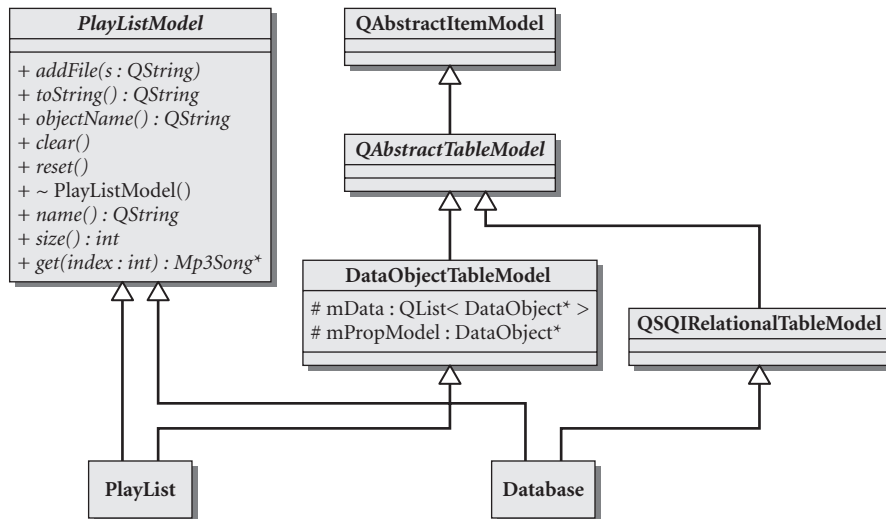


FIGURE 25.8 `PlayListTableModel`

To model a collection of `DataObjects` as a table, you can reuse `DataObjectTableModel`. This class determines what to display in each column based on Qt properties. This makes `PlayListView` much easier to write.

1. Write a `PlayListView` class. You do not need to write any code for databases, but keep the design considerations of Figure 25.8 in mind as you write it.
2. Enhance the GUI you wrote in Section 25.8, by adding a `load playlist` button. Add a `QAction`-derived class called `LoadPlayListAction`. Write the action so that it fills up the contents of the `PlayListView`.
3. Make the `Mp3Player` work so that it can automatically play one song after another in the loaded playlist.

25.10 Source Selector

Being able to select from a variety of different “sources” is what makes a player very powerful and useful. A source could be any of the following:

- A playlist
- A database
- A list of online radio stations
- Tracks on a CD

Any of these can be considered an input source. The other interesting thing they all have in common is that they can all be represented by a `PlaylistModel` (a collection of songs).

The `SourceSelector` widget, shown docked on the left in Figure 25.9, permits the user to select the currently playing “source,” which in this case corresponds to a `PlaylistModel`. However, the `SourceSelector` manages a mapping of models to views, to make switching views easy and fast.

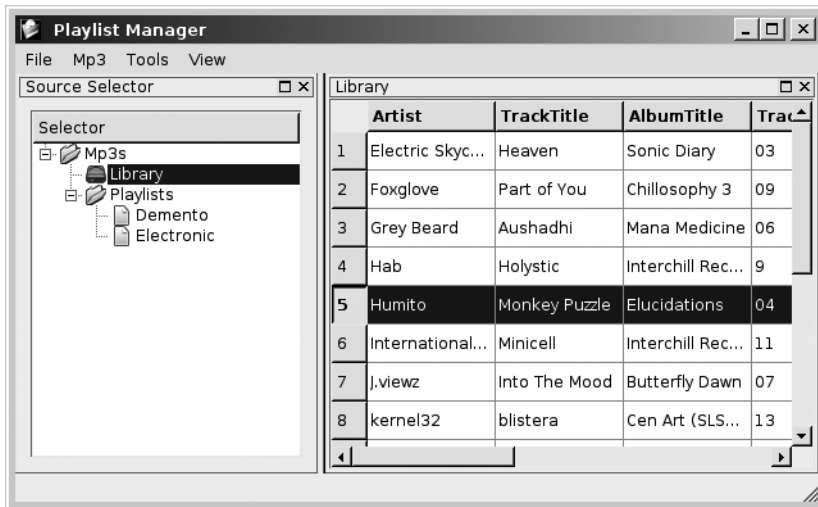


FIGURE 25.9 Source Selector view

Clicking on one of the sources in the selector tree should change the currently visible table or list view in the central widget. Figure 25.10 shows one possible way of designing the classes that provides a view and a selector of a collection of sources.

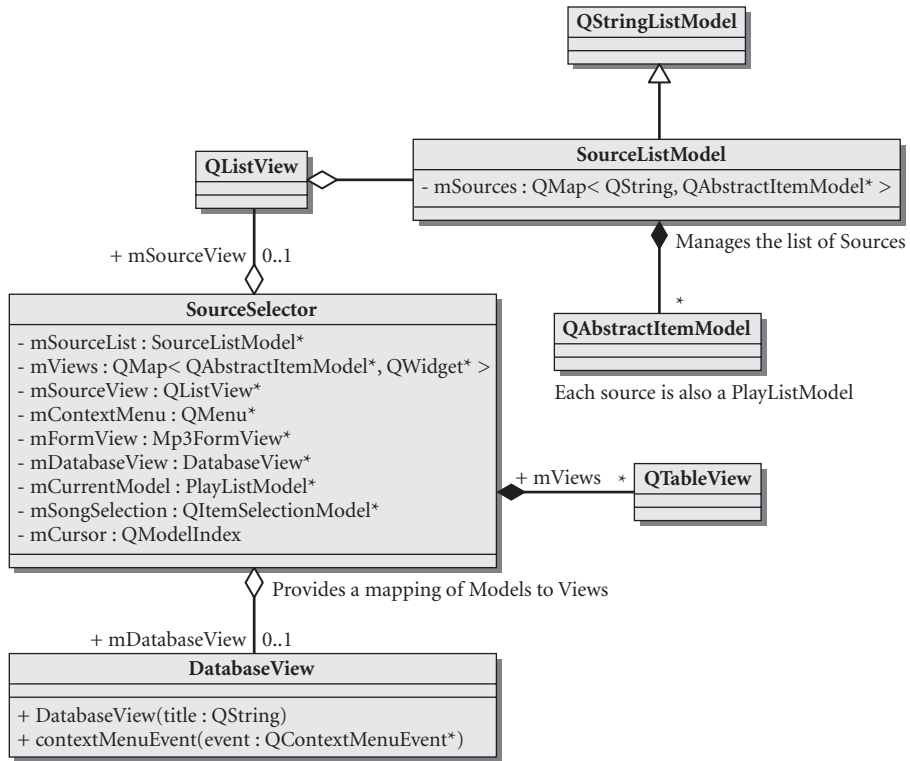


FIGURE 25.10 SourceSelector and related classes

25.11 Persistent Settings



QSettings (Section 11.2.1)

Suppose you want to be able to support multiple playlist formats and also remember the last format used. Or, suppose you want to remember the last opened directory for MP3 files and, independently, the last opened directory for playlist files.

Settings are simply a persistent mapping of name-value pairs. The names can be anything we like, but using meaningful hierarchical names will aid greatly in organization. In Qt on *nix, people tend to use the slash (“/”) as a namespace delimiter—in contrast to Java, which uses a dot (“.”).

In this exercise, we will implement persistent settings by reusing QSettings. A QSettings object reads/stores its settings from/to different places depending on the operating system.

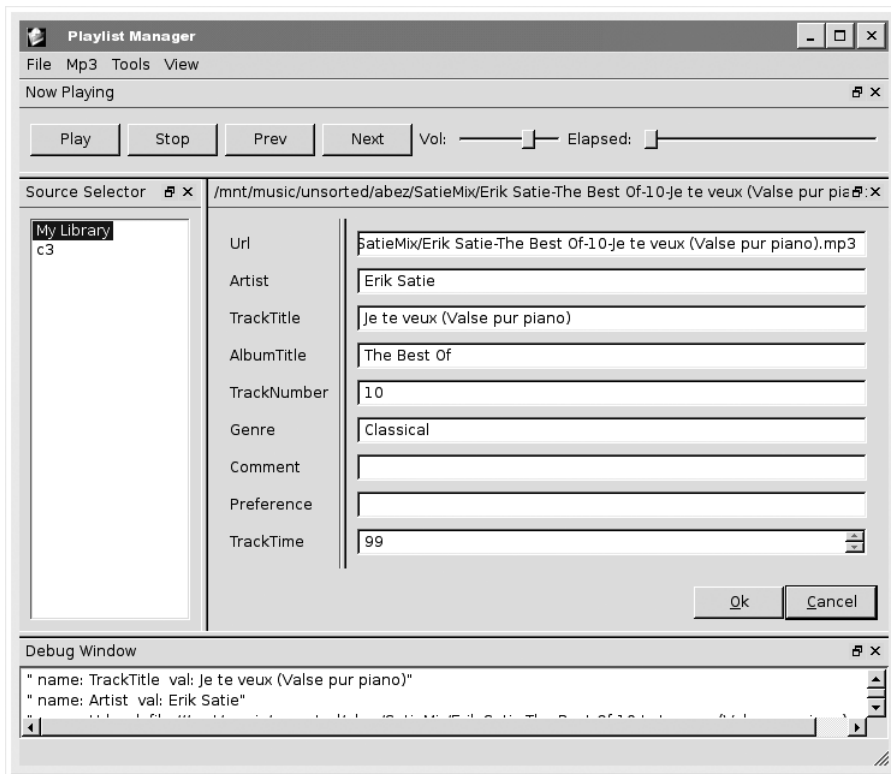
The first three settings that the program needs to remember across executions are listed below. Write a `TestCase` of the settings class that loads, changes, saves, loads again, and checks whether the values were changed.

- `playlistmgr/outputformat=(extm3u|xml)`
- `playlistmgr/lastopendir=last opened directory`
- `playlistmgr/playlistdir=last opened playlist directory`

Think of other properties/settings you might want to set from the application that must persist across executions.

25.12 Edit Form View for FileTagger

Whenever an MP3 song is selected, we want to be able to execute an action on it: edit tag. This choice should go in context menus for all playlist views, as well as in the MP3 pull-down menu.



Write a form view for `FileTagger`, called `Mp3TagForm`, that provides the user with editable, viewable information about all of the song's properties. We need

action buttons as follows:

- **Ok:** Perform a `commit()` on the `FileTagger` (which stores changes to the actual `id3tag` in your MP3 file) and return to main screen (hide form).
- **Reload:** Discard changes and reload tag info.
- **Cancel:** Discard changes and return to main screen (hide form).
- **Play:** Play the track.

When editing a song, it should use a `QLineEdit` for each `QString` field and a `QComboBox` for the preference values. Keep in mind that the URL, filename, and track length are non-editable fields. Finally, you can use Designer or hand-code your dialog.

25.13 Database View

A **database view** is not very different from an ordinary playlist view except that it permits you to see a larger collection of songs, organized in different ways to aid in the selection process.

Typically, a database view can be achieved with a table, a tree of lists, or a table of lists. The simplest way is to use the `QTableView` and then connect it to a model that represents the data.

Qt, when it is built with the correct options, can access SQL databases, so that we can use `QSqlTableModel`. Using this with the regular `QTableView` gives you an editable tabular view of a database table. The database view shown in Figure 25.11 looks like an ordinary table view, because it is! It's just hooked up to a database model.

	Artist	TrackTitle	AlbumTitle	Trac	Genre	Preference	Filename
37	Limborg	Exode	siorapalouk	08	World Fusion	Very Good	/mnt/music/DJ Oddiofile Compila...
38	Limborg	Shakita	siorapalouk	11	World Fusion	Very Good	/mnt/music/DJ Oddiofile Compila...
39	Manic P.	Lost in Silence	Heavenly Voices 1	13	4AD	Very Good	/mnt/music/DJ Oddiofile Compila...
40	Mari Boine	Gula Gula [C...	Cafe del Mar, Vol. 8	05	Electronica	Very Good	/mnt/music/DJ Oddiofile Compila...
41	Marta Sebes...	Sino Moi	Kismet	02	World Fusion	Very Good	/mnt/music/DJ Oddiofile Compila...
42	Pois Z'ont Ro...	la java des m...	le Bal des Baleines	2	French	Very Good	/mnt/music/DJ Oddiofile Compila...
43	Sanna Kurki...	Minne - Where	Musta	03	World Fusion	Very Good	/mnt/music/DJ Oddiofile Compila...
44	Susanne Lun...	Havella (Old ...	Wizard Women of t...	04	World	Excellent	/mnt/music/DJ Oddiofile Compila...
45	Tallari	Miksi Ne Neij...	Wizard Women of t...	03	World	Excellent	/mnt/music/DJ Oddiofile Compila...
46	Taras Bulba	the truth	Basta Cosi	02	Techno	Very Good	/mnt/music/DJ Oddiofile Compila...
47	Taras Bulba	Der Wahrheit...	Peyote Moon	09	Chill	Very Good	/mnt/music/DJ Oddiofile Compila...
48	Taras Bulba	Barune (Myst...	Sketches of Babel	02	4AD	Very Good	/mnt/music/DJ Oddiofile Compila...

FIGURE 25.11 QTableView of a database

EXAMPLE 25.5 `../src/libs/filetagger/filetagger.sql`

```

create database mp3db;
use mp3db;
grant all on mp3db.* to 'mp3db'@'localhost' identified by 'mp3dbpw';
grant all on mp3db.* to mp3db identified by 'mp3dbpw';

drop table FileTagger;
CREATE TABLE FileTagger (
  Artist      varchar(100),
  TrackTitle  varchar(100),
  AlbumTitle  varchar(100),
  TrackNumber varchar(10),
  Genre       varchar(20),
  Comment     varchar(200),
  Preference  varchar(20),
  Filename    varchar(200),
  PRIMARY KEY(Filename),
  INDEX(Preference),
  INDEX(Artist),
  INDEX(Genre)
);

```

Example 25.5 contains the SQL table definition, and Figure 25.12 contains a suggested initial design for the model.

1. Extend the `Mp3TableModel` so that it derives from `PlaylistModel` and fits into your larger program.
2. Write a Library menu with an option to “import songs into library,” which imports the songs into the database and updates the view.

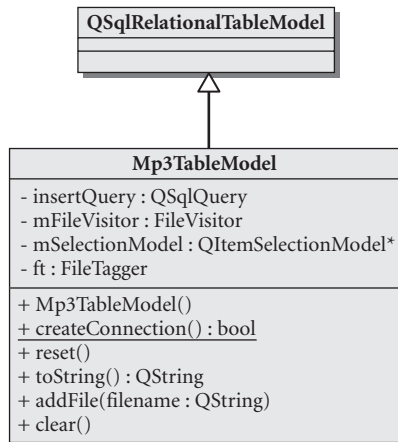


FIGURE 25.12 Database table model

POINTS OF DEPARTURE

If you reached this point, you have a strong understanding of the benefits and difficulties of model-view programming. If you compare your application to something like `juK` or `amaroK`, you can come up with ideas for other features to add. This might be a good time to download and examine source code of these and other existing KDE projects—you will notice most of them are Qt 3 based, and you should be able to understand the code now. If you feel confident enough, you might want to help in the maintenance or porting of existing KDE projects over to Qt 4 / KDE4. Join the mailing lists for your favorite projects.



P A R T V
Appendices



Appendix A. C++ Reserved Keywords	575
Appendix B. Standard Headers	577
Appendix C. The Development Environment	579

Appendix A: C++ Reserved Keywords

Keywords are identifiers that are part of the basic syntax of the language. These names have fixed meanings and cannot be used in any way that attempts to change those meanings.

Here is a list of keywords in C++. Those that are shown in **bold** are also part of ANSI C89.

and	dynamic_cast	operator
and_eq	else	or
asm	enum	or_eq
auto	explicit	private
bitand	export	protected
bitor	extern	public
bool	false	register
break	float	reinterpret_cast
case	for	return
catch	friend	short
char	goto	signed
class	if	sizeof
compl	inline	static
const	int	static_cast
const_cast	long	struct
continue	mutable	switch
default	namespace	template
delete	new	this
do	not	throw
double	not_eq	true

try	unsigned	wchar_t
typedef	using	while
typeid	virtual	xor
typename	void	xor_eq
union	volatile	

Appendix B: Standard Headers

This book uses a small subset of the **Standard Template Library** (STL; also called the **Standard Library**). The standard header files we use are all listed here.

To use these classes and functions effectively, it is useful to know where to look for documentation.

Table B.1 lists standard header files. For some header files, there is a `man` page for the whole file. In other cases, you might find a `man` page for the individual function also.

If you are using Microsoft Developer's Studio, the documentation for the standard libraries comes with the MSDN documentation.

For open-source platforms, it helps to have one but you don't need a local copy of `man` or the `man` pages since there are many copies of the documentation¹ available on the Web.

¹ For examples, see `cplusplus.com` [<http://www.cplusplus.com/ref/>] or `Dinkumware` [<http://www.dinkumware.com/manuals/reader.aspx?lib=cpp>].

TABLE B.1 Standard headers

Header file	Library	Man pages
C++ STL		
string	STL strings type	std::string
sstream	stringstream, for writing to strings as if they are streams	std::stringstream
iostream	C++ standard stream library	std::ios, std::iostream
memory	C++ memory-related routines	std::bad_alloc, std::auto_ptr
C Standard Library		
cstring, string.h	functions for C char* strings	string, strcpy, strcmp
cstdlib, stdlib.h	c Standard Library	random, srandom, getenv, setenv
cstdio, stdio.h	standard input/output	stdin, stdout, printf, scanf
cassert	assert macros	assert



By default, the C++ Standard Library documentation might not be installed on your system. Search for the string libstdc with your favorite package manager, so you can install something like libstdc++5-3.3-doc or libstdc++6-4.0-doc.

Appendix C: The Development Environment

C.1 The Preprocessor: For #including Files

This appendix explains some of the mysteries of the C preprocessor, class declarations versus including headers, and some best practices to reduce dependencies between header files.

In C++, code reuse is indicated by the presence of a preprocessor directive, `#include`, in source code and header files. We `#include` header files that contain things like class or namespace definitions, `const` definitions, function prototypes, and so forth. These files are literally *included* in our own files before the compiler begins to translate our code.

The compiler will report an error if it sees any identifier defined more than once. It will tolerate repeated declarations but not repeated definitions.¹ To prevent repeated definitions, we are always careful to use an `#ifndef` *wrapper* around each header file. This tells the C preprocessor to skip the contents if it has already seen them. Let's examine the following class definition in Example C.1.

¹ We discuss the difference between declaration and definition in Section 20.1.

EXAMPLE C.1 `src/preprocessor/constraintmap.h`

```

#ifndef CONSTRAINTMAP_H
#define CONSTRAINTMAP_H

/* included class definitions: */
#include <QHash>
#include <QString>

class Constraint; ❶

class ConstraintMap : public QHash<QString, Constraint*> { ❷
private:
    Constraint* m_Constraintptr; ❸
    // Constraint m_ConstraintObj; ❹
    void addConstraint(Constraint& c);
};

#endif // #ifndef CONSTRAINTMAP_H

```

- ❶ a forward declaration
 - ❷ Needs definitions of QHash and QString, but only the declaration of Constraint, because it's a pointer.
 - ❸ No problem—it's just a pointer.
 - ❹ error—incomplete type
-

As you can see, as long as we use pointers or references, a forward declaration will suffice. The pointer dereferencing and member accessing operations are performed in the implementation file, which needs the full definition of all types it uses.

EXAMPLE C.2 `src/preprocessor/including.cpp`

```

#include "constraintmap.h"

ConstraintMap map; ❶

/* redundant but harmless if #ifndef wrapped */
#include "constraintmap.h"

// Constraint p; ❷
#include <constraint.h>
Constraint q; ❸

```

- ❶ Okay—ConstraintMap already included.
 - ❷ error—incomplete type
 - ❸ Now it is a complete type.
-

Here are some guidelines to help decide whether you need a forward declaration or the full header file to `#include` in your header file:

- If *ClassA* derives from *ClassB*, the definition of *ClassB* must be known by the compiler when it processes the definition of *ClassA*. Therefore, the header file for *ClassA* must `to` the header file for *ClassB*.
- If the definition of *ClassA* contains a member that is an object of *ClassD*, the header file for *ClassA* must `#include` the header file for *ClassD*. If the definition of *ClassA* contains a function that has a parameter or a return object of *ClassD*, the header file for *ClassA* must `#include` the header file for *ClassD*.
- If the definition of *ClassA* only contains non-dereferenced *ClassE* pointers, then a **forward declaration** of *ClassE* is sufficient in the *ClassA* header file:

```
class ClassE;
```

A class that is *declared* but not *defined* is considered an **incomplete type**. Any attempt to dereference a pointer or define an object of an incomplete type will result in a compiler error.²

The implementation file, `classa.cpp`, for *ClassA* should `#include` "`classa.h`" and also `#include` the header file for each class that is used by *ClassA* (unless that header file has already been included in `classa.h`). All pointer dereferencing should be performed in the `.cpp` file. This helps reduce dependencies between classes and improves compilation speed.

A `.cpp` file should never `#include` another `.cpp` file. A header file should `#include` as few other header files as possible so that it can be included more quickly and with fewer dependencies. A header file should always be `#ifndef` wrapped to prevent it from being included more than once.

² The actual error message may not always be clear, and with `QObject`s, it might come from the MOC-generated code rather than your own code.

Circular Dependencies

Whenever one file `#includes` another, there is a strong **dependency** created between the files. When a dependency like this exists between header files, it cannot be bidirectional: The preprocessor is unable to cope with a **circular dependency** between header files, where each one `#includes` the other. One of the `#include` statements must be replaced by a forward class declaration.

Forward declarations help remove circular dependencies between classes and, in the process, enable bidirectional relationships to exist between them.

C.2 Understanding the Linker

Figure C.1 shows how the linker accepts binary files, which were generated by the compiler, and creates executable binaries as its output. The linker executable, on *nix machines is simply called `ld`, and it is run by `g++` after all source files are compiled successfully.

All of these steps are performed when you run `make`, which prints out every command before it executes. By reading the output of `make`, you can see what arguments are being passed to the compiler and linker. When an error occurs, it immediately follows the command line that produced the error.

Example C.3 shows the command line options passed to `g++`, and attempts to show how `g++` runs the linker, known as `ld`, and also passes some arguments to it.

EXAMPLE C.3 linker-invocation.txt

```
g++ -Wl,-rpath,/usr/local/qt-x11-free-3.2.3/lib           ❶
-o hw7                                                  ❷
.obj/address.o .obj/ca_address.o .obj/constraintgroup.o
.obj/customer.o .obj/dataobject.o .obj/dataobjectfactory.o
.obj/hw07-demo.o .obj/us_address.o .obj/moc_address.o   ❸
.obj/moc_ca_address.o .obj/moc_customer.o .obj/moc_dataobject.o
.obj/moc_us_address.o
-L/usr/local/qt-x11-free-3.2.3/lib -L/usr/X11R6/lib
-L/usr/local/Utils/lib                                  ❹
-lutils -lqt -lXext -lX11 -lm                          ❺
```

- ❶ Tells g++ to run the linker, and pass these options to ld.
- ❷ Specify the output to be called hw7.
- ❸ Link these object files into the executable.
- ❹ Add another location for the linker to search for libraries.
- ❺ Link this app with five more libraries: qt utils, ext, X11, and m

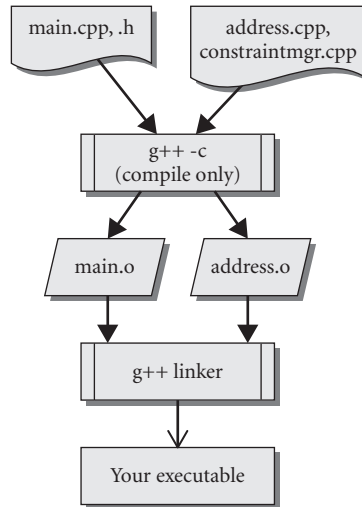


FIGURE C.1 The linker's inputs and outputs

Linking entails the following:

- For each library name passed with the `-l` switch, find the corresponding library file, searching the library path, as well as all `-L` switched arguments (which were generated by `qmake` from the `LIB` `qmake` variable).
 - For static libraries, it contains the code to be linked into the executable.
 - For dynamic libraries, it is a catalog listing (often in readable ascii format) that describes where the actual shared objects are for each label definition. The linker will check to make sure the shared object is where it should be, and report an error if not.
- For each function that is called from any place in the code we are linking, find the object where that code is located and do a simple, fast check to determine that there is, indeed, a completely defined function with the

proper name/signature at that location. Report an error if it can't be found or isn't the correct type/name/size.

- For each reference to a variable name, find the object-address where that variable is located and do a simple, fast check to make sure the address is a valid one for an object of that type.

This is the general idea. The linker resolves references to names by finding their real addresses in files and checking the addresses to determine whether they're valid for the type id. It's like a directory look-up service for C++ compilers.

C.2.1 Common Linker Error Messages

C++ programmers sometimes spend lots of time trying to understand and repair compiler and linker errors. If you can't understand the message, you're stuck. With a compiler error, the problem is easier to diagnose because it is related to the compilation of one source code module and the header files it includes. The compiler generally tells you the exact location of any error that it detects. With a linker error, the problem is related to how your source code modules link together. When the linker stage is reached, all the individual modules have compiled without errors. Linker errors can be caused by bugs in C++ code, but they can also be a result of mistakes in the project file.

C.2.1.1 Unable to Find libxxx.so.x

For Win32 Users

At build-time, your IDE needs to be able to find the .DLL. To configure it, drill into your menu structure until you find `project -> properties -> c/c++ build -> libraries`. Here you can *add* a third-party library, and you'll be asked in a dialog for the location of headers and DLL files.

At runtime, your PATH system environment variable must contain the directory where the required DLLs are located.

Installing a library means making it available for more than a single user on a system. It is also possible to reuse a library without installing it. All libraries that you reuse must either be installed or placed in a directory listed in your LD_LIBRARY_PATH.

When you are reusing a library for the first time, you will probably see this error message. It means that the linker cannot find the library. When the gnu linker looks for a shared object, it checks at least two places:

1. The directories specified in `LD_LIBRARY_PATH`
2. Installed libraries referenced from a cache file called `/etc/ld.so.cache`

The Cache File `ld.so.cache`

The cache file provides fast lookup of shared objects found in the directories specified in `/etc/ld.so.conf`. Some directories you might find there are

```
/lib
/usr/lib
/usr/X11R6/lib
/usr/i486-linuxlibc1/lib
/usr/local/lib
/usr/lib/mozilla
```

If you use a Linux package installer to install a library, it will probably make the proper changes to `ld.so.conf` and rebuild your cache file. However, if you manually compile and install libraries, it may be necessary for you to edit this file. Afterwards, you can rebuild the cache file with the command `ldconfig`.

C.2.1.2 Undefined Reference to [identifier]

This is the most common and probably the most annoying linker error of all. It means that the linker cannot find the definition of some named entity in your code. Here is some output from `make`.

```
.obj/ca_address.o(.gnu.linkonce.t._ZN10DataObject16getConstraint-
MgrEv+0x4):
In function 'DataObject::getConstraintMgr()':
/usr/local/qt-x11-free-3.2.3/include/qshared.h:50:
undefined reference to 'DataObject::sm_Cm'
collect2: ld returned 1 exit status
make: *** [hw7] Error 1
```

The compiler found the declaration, but the linker can't find the corresponding definition. In some part of your code, you are *referencing* a symbol, but there is no definition found. The useful bits of information are

- The symbol it can't find is `DataObject::sm_Cm`.
- The function that is trying to use it is
`DataObject::getConstraintMgr`.

The first step is to determine whether we, as humans, can find the missing definition. If we can't, how can the linker? If we find it in a `.cpp` file, we must make sure that

- Both the `.cpp` and the `.h` file are mentioned in the project
- The file is included in a library with which we are linking

Because we are using good naming conventions (see Section 3.4), we can immediately tell that `sm_Cm` is a static data member of class `DataObject`. The compiler found the declaration, but the linker can't find the definition.

Because it is static (Section 2.10), the definition for `sm_Cm` belongs in `dataobject.cpp`. The compiler expects to find a definition statement of the form:

```
ConstraintMgr DataObject::sm_Cm;
```

If it's there and the linker still can't find it, the most likely causes for this error are

- The `.cpp` file that contains the definition is not listed in `qmake's SOURCES` in the `.project` file.
- The code is located in another library but the linker can't find the library. This is solved by adding a missing `LIBS` argument in the project file.
 - `-lmylib` adds a library to be linked.
 - `-Lmylibdir` adds a directory to the linker's lib search path list.

C.2.1.3 Undefined Reference to vtable for ClassName

This is one of the most confusing errors. It generally means that a virtual function definition is missing. Literally, the vtable for that class (which has addresses of each the virtual functions) is unable to be fully constructed.

This error can arise from missing function definitions in your code, but it can also be caused by a missing `HEADERS` or `SOURCES` entry in your `make/project` file. The resolution is to double-check that all files are listed properly in the project file before you delve too deeply into your C++ code.

All-inline Classes

For polymorphic classes,³ there should be at least one non-inline definition (function or static member) in a source (`.cpp`) file for that header file. *Without this, many linkers will not be able to find any of its `virtual` method definitions.*

All-inline classes are legal in C++, but they do not work in their intended way when mixed with polymorphism.

C.3 Debugging

The compiler can locate and describe syntax errors. The linker can reveal the existence of inconsistencies among program components and give some help as to how to locate them. One of the most challenging aspects to using C++ is learning how to find and fix various kinds of run-time errors.

Run-time errors are logical errors that can exist in a program that is syntactically correct and contains no undefined objects or functions. Effective use of a debugger, a program specifically designed for tracking down runtime errors, can greatly reduce the amount of time spent dealing with these kinds of errors.

A debugger permits the stepwise execution of your code, as well as the inspection of object values. Since debuggers work with compiled code, the early versions could only be used by programmers who were familiar with assembly language. Modern debuggers are able to step concurrently through the compiled machine code and the original source code. The GNU family of developer tools includes `gdb`, the source-level GNU debugger, which we can use for C/C++ applications. `gdb` has been designed with a command-line interface that is quite powerful but not particularly user-friendly. Fortunately, there are several open-source graphical facades for `gdb`, one of which we will discuss below. Commercial C++ IDEs (e.g., Visual Studio) generally have built-in source-level debuggers.

³ classes with at least one `virtual` method

C.3.1 Building a Debuggable Target

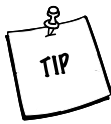
For `gdb` to work, debugging symbols must be built into the code at compile time. Otherwise, the machine instructions will not be mapped to locations in C++ source files. This is easily accomplished by using the appropriate command-line switch (`-g`) when invoking the compiler:

```
g++ -g filename.cpp
```

This often results in a *significantly* larger executable file. Generally, the growth is proportional to the size and complexity of the source code files. The expanded executable contains symbol table information that the debugger can use to find source code that corresponds to machine instructions. To get `qmake` to generate makefiles with the `-g` switch passed along to `g++`, add the following line to your `qmake` project file:

```
CONFIG += debug
```

When the Qt library has been built with debugging symbols, you can step through the Qt source code just as easily as your own code. You may need to build Qt with debugging symbols to debug certain programs that contain code directly called from the Qt library.



BUILDING QT WITH DEBUGGING SYMBOLS In Win32, it's a menu choice you can click on. On *nix platforms, after unpacking the source code tarball, pass a parameter to the `configure` script before building and your Qt library will be built with debug symbols.

```
./configure --enable-debug  
make  
make install
```

EXERCISE: BUILDING A DEBUGGABLE TARGET

- Compare the size of an executable file created with and without the `CONFIG += debug` line in the project file.
- Make a mental note to try this again later with a more complex application.

C.3.2 gdb Quickstart

Imagine you are running a program and, for some mysterious reason, it crashes.

```
[lazarus] app> ./playlistmgr
Segmentation fault
[lazarus] app>
```

When your app aborts, or crashes, it is helpful to know (as quickly as possible) exactly where it happened. We can use `gdb` to locate the trouble spot quickly and easily.

```
[lazarus] app> gdb playlistmgr
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
This GDB was configured as "i386-linux"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb) r4
Starting program: ftgui/app/playlistmgr
[Thread debugging using libthread_db enabled]
[New Thread -1227622176 (LWP 17021)]
Qt: gdb: -nograb added to command-line options.
        Use the -dograb option to enforce grabbing.
This is a debug message

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1227622176 (LWP 17021)]
0xb7f03320 in FormDialog::createActions (this=0x80ae2a0) at
formdialog.cpp:53
53         delete m_OkAction;
(gdb)
```

`gdb` shows you not only the filename, and line number, but also the corresponding line in the source code. However, we still might want to get some context for this error. The command `list` shows you the surrounding source code for the current file:

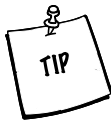
```
(gdb) list
51     void FormDialog::createActions() {
52
53         delete m_OkAction;
54         delete m_CancelAction;
55         m_OkAction = new OkAction(m_Model, m_View);
56         m_CancelAction = new CancelAction(m_Model, m_View);
57         QHBoxLayout *buttons = new QHBoxLayout(0);
(gdb)
```

⁴ `r` is the command for “run.”

The command `where` shows you the stack trace, or how we got there.

```
(gdb) where
#0  0xb7f03320 in FormDialog::createActions (this=0x80ae2a0) at
formdialog.cpp:53
#1  0xb7f03058 in FormDialog::setModel (this=0x80ae2a0,
fmodel=0x80c80d0)
    at formdialog.cpp:34
#2  0x080664bd in SettingsDialog (this=0x80ae2a0, parent=0x0) at
settingsdialog.cpp:14
#3  0x0805f313 in MainWindow (this=0xbffffdec8) at
mainwindow.cpp:42
#4  0x08066f14 in Controller (this=0xbffffdec0, argc=1,
argv=0xbffffdf4) at controller.cpp:25
#5  0x0805a8a4 in main (argc=1, argv=0xbffffdf4) at main.cpp:7
(gdb)
```

Most open-source IDEs use `gdb` under the hood. They each offer a user interface that makes certain features easier to learn and use. Four open-source apps that provide a front-end for `gdb` are: `Eclipse`, `kdevelop`, `kdbg`, and `ddd`.



VIEWING QSTRINGS INSIDE THE DEBUGGER `QString`s are hard to see inside some debuggers because they are indirect pointers to Unicode data. The debugger needs to know extra things about a `QString` in order to display it properly.

Download these Qt 4 helper macros from the KDE subversion repository⁵ and put this in your `~/ .gdbinit`:

```
source /path/to/kde/kde-devel-gdb
define pqs
    printq4string $arg0
end
```

Now you should be able to print `qstrings` with the `pqs` macro.

C.3.3 Finding Memory Errors

Memory errors are very difficult to track down without the aid of a run-time analysis tool. A program that analyzes the running performance of a program is called a **profiler**. `valgrind` is an open-source profiling tool for Linux that tracks

⁵ http://websvn.kde.org/*checkout*/trunk/KDE/kdesdk/scripts/kde-devel-gdb

the memory and CPU usage of your code and detects a variety of errors. These include

- Memory leaks—memory that is no longer accessible but which has not been deleted
- Invalid pointer use for heap memory, such as
 - Out of bounds index
 - Mismatches between allocation and deallocation syntax (e.g., allocating with `new[]` but deallocating with `delete`)
- Use of uninitialized memory

Each of the errors just listed can cause catastrophic results in a piece of software. Profilers can also be used for performance tuning and determining which code is responsible for slowing down a program (i.e., finding bottlenecks).

Example C.4 shows a short program that contains a deliberate memory usage error.

EXAMPLE C.4 `src/debugging/wrongdelete.cpp`

```
void badpointer1(int* ip, int n) {
    ip = new int[n];
    delete ip; ❶
}

int main() {
    int* iptr;
    int num(4);
    badpointer1(iptr, num);
}
```

❶ wrong delete syntax

For the output to be human readable, we compile with debugging symbols (-g).

```
debugging/wrongdelete> g++ -g -pedantic -Wall wrongdelete.cpp
debugging/wrongdelete> ./a.out
debugging/wrongdelete>
```

The compiler didn't complain, and even after running the program, no error behavior is exhibited. However, memory is corrupted by this program.

Here is a (slightly abbreviated) look at valgrind's analysis of our program. We have removed the process id of the valgrind job from the beginning of each line. The process id will, of course, be different each time you run valgrind.

```
src/debugging> valgrind a.out
--3332-- DWARF2 CFI reader: unhandled CFI instruction 0:50
--3332-- DWARF2 CFI reader: unhandled CFI instruction 0:50
Mismatched free() / delete / delete []
  at 0x401C1CB: operator delete(void*) (vg_replace_malloc.c:246)
  by 0x80484BD: badpointer1(int*, int) (wrongdelete.cpp:3)
  by 0x80484F4: main (wrongdelete.cpp:9)
Address 0x4277028 is 0 bytes inside a block of size 16 alloc'd
  at 0x401BBF4: operator new[](unsigned)
  (vg_replace_malloc.c:197)
  by 0x80484AC: badpointer1(int*, int) (wrongdelete.cpp:2)
  by 0x80484F4: main (wrongdelete.cpp:9)
```

valgrind found the errors and, with debugging symbols, could point us to the location of the problem code. Example C.5 is a little more interesting because it contains memory leaks and array index errors.

EXAMPLE C.5 src/debugging/valgrind-test.cpp

```
#include <iostream>

int badpointer2(int k) {
    int* ip = new int[3];
    ip[0] = k;
    return ip[3];    ❶
}                  ❷

int main() {
    using namespace std;
    int* iptr;
    int num(4), k;    ❸
    /* what is the state of iptr? */
    cout << iptr[num-1] << endl;
    cout << badpointer2(k) << endl;
}
```

- ❶ out of bounds index
 - ❷ memory leak
 - ❸ k is uninitialized.
-

Running Example C.5 through `valgrind` shows us the exact locations of some errors.

For more details, rerun with: `-v`

```
--2164-- DWARF2 CFI reader: unhandled CFI instruction 0:50
--2164-- DWARF2 CFI reader: unhandled CFI instruction 0:50
Use of uninitialised value of size 4
   at 0x80486AF: main (valgrind-test.cpp:17)
68500558

Invalid read of size 4
   at 0x804867C: badpointer2(int) (valgrind-test.cpp:8)
   by 0x80486DD: main (valgrind-test.cpp:18)
Address 0x4277034 is 0 bytes after a block of size 12 alloc'd
   at 0x401BBF4: operator new[](unsigned) (vg_replace_malloc.c:197)
   by 0x8048667: badpointer2(int) (valgrind-test.cpp:6)
   by 0x80486DD: main (valgrind-test.cpp:18)
```

0

```
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 19 from 1)
malloc/free: in use at exit: 12 bytes in 1 blocks.
malloc/free: 1 allocs, 0 frees, 12 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 1 not-freed blocks.
checked 120,048 bytes.
```

LEAK SUMMARY:

```
definitely lost: 12 bytes in 1 blocks.
possibly lost: 0 bytes in 0 blocks.
still reachable: 0 bytes in 0 blocks.
suppressed: 0 bytes in 0 blocks.
```

Use `--leak-check=full` to see details of leaked memory.

If this is not enough information to find where the memory leak is, we can rerun `valgrind` with the switch `--leak-check=full`.

C.4 Qt Assistant and Designer

Qt comes with two developer's tools: `assistant` and `designer`.

`Assistant` is an extensible help browser, similar to a Web browser, but it supports various built-in search and index capabilities.

Designer not only showcases most of Qt's widgets, it also permits you to create and lay out customized dialogs and widgets. After you are finished designing your UI, Designer can write the following files for you:

<i>widgetName.ui</i>	An xml file that represents a tree of objects and properties. <code>uic</code> , or the "UI compiler," comes with Qt and generates C++ code from this xml file. <code>uic</code> gets run automatically from <code>make</code> , because <code>qmake</code> generates a <code>Makefile</code> that runs <code>uic</code> on all <code>.ui</code> files in the project.
<i>projectName.pro</i>	Designer can open and manipulate <code>qmake</code> project files, adding forms to the projects.
<i>widgetName.ui.h</i>	C++ source code that is meant to be inserted into generated code by <code>uic</code> .

Qt Assistant provides tutorials for using the latest version of Qt Designer.

C.5 Open-Source IDEs and Development Tools

It is not practical to do object-oriented development with an ordinary text editor. Object-oriented development involves working with many classes and many more files (headers and sources). Writing code in an edit window is just a small part of the development process. A good programmer's editor or **IDE (integrated development environment)** should support many of the following features:

- Tree-like structured navigation to object/members in any file
- Refactoring assistance for moving/renaming members
- Integrated debugger
- Context-sensitive help linked to API documentation
- A built-in command-line shell window so you can run programs without leaving your environment
- A project manager to help manage groups and subgroups of related files
- Editing modes in other programming languages
- Easy keyboard customization—the ability to make any keystroke perform any task (cursor movement especially, but also window movement)
- An open plug-in architecture so you can add other components
- Integration with a version-control facility is desirable, especially in windows environments. Look for CVS,⁶ Subversion,⁷ or Darcs⁸.

⁶ <https://www.cvshome.org/>

⁷ <http://subversion.tigris.org/>

⁸ <http://darcs.net/>

- Learnable, scriptable macros
- Easy language-aware navigation to your different files (with shortcuts such as “find declaration,” “find definition,” and “find references”)

An open-source option for Win32 users is Dev C++⁹ from Bloodshed Software, which works quite well with MinGW and cygwin.

KDE users can use KDevelop³,¹⁰ a feature-rich, open-source IDE with excellent C++ and code navigation features. It has built-in support for importing Qt’s `qmake` project files. Simply select *Project -> Import Existing Project* from the menu and choose the `.pro` file you wish to work with.

For all platforms, there is Eclipse,¹¹ a free Java-based open-source IDE. You can download plugins for C++ development,¹² as well as Qt/KDE development.¹³ The latter allows you to import `qmake .pro` files into Eclipse as projects directly.

Maximum Code Reuse: KDevelop

An interesting thing about KDevelop is that it embeds stand-alone applications and “plugs them in” to the `QMainWindow` as dock widgets. If you already use KDE and some of the common *nix development tools, you will find some familiar apps already available inside the dock windows. In the case of KDevelop:

- The debugger is a KDE front end to `gdb`, similar to `kdbg`.
- For CVS browsing, KDevelop has integrated `cervisia`.
- By default, KDevelop uses Kate¹⁴ (the “KDE Advanced Text Editor”) for editing.

(continued)

⁹ <http://www.bloodshed.net/devcpp.html>

¹⁰ <http://www.kdevelop.org>

¹¹ <http://www.eclipse.org>

¹² <http://www.eclipse.org/cdt/>

¹³ <http://kde-eclipse.pwsp.net/index.php>

¹⁴ <http://kate.kde.org/>

- The designer for creating widgets is a customized version of Qt's Designer.
- The command-line shell is a dockable window inside KDevelop. It's the regular KDE Konsole¹⁵ xterminal.

Because KDevelop is built on top of KDE libraries, and KDE libraries are based on Qt, using KDevelop will enable you to become more accustomed to how various Qt widgets work. See Figure C.2.

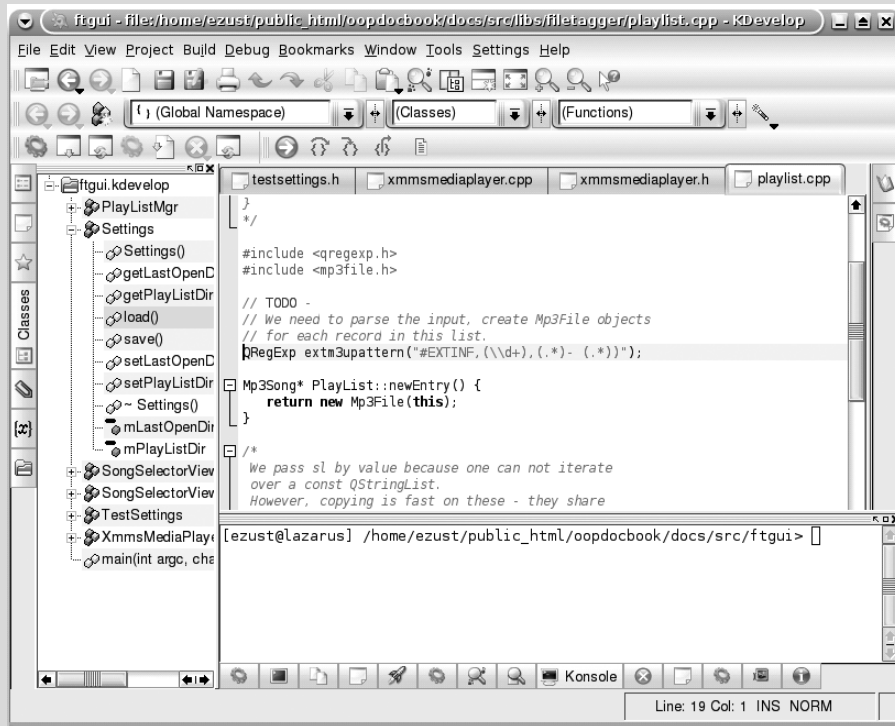


FIGURE C.2 KDevelop

¹⁵ <http://konsole.kde.org/>

C.5.1 UML Modeling Tools

For creating diagrams in this book using the Unified Modeling Language, we use two open-source tools, Umbrello¹⁶ and Dia.¹⁷ Each tool uses an XML dialect as its native file format.

Umbrello is the KDE UML Modeler. It can directly import C++ code, making it very easy to drag and drop imported classes into diagrams, as shown in Figure C.3.

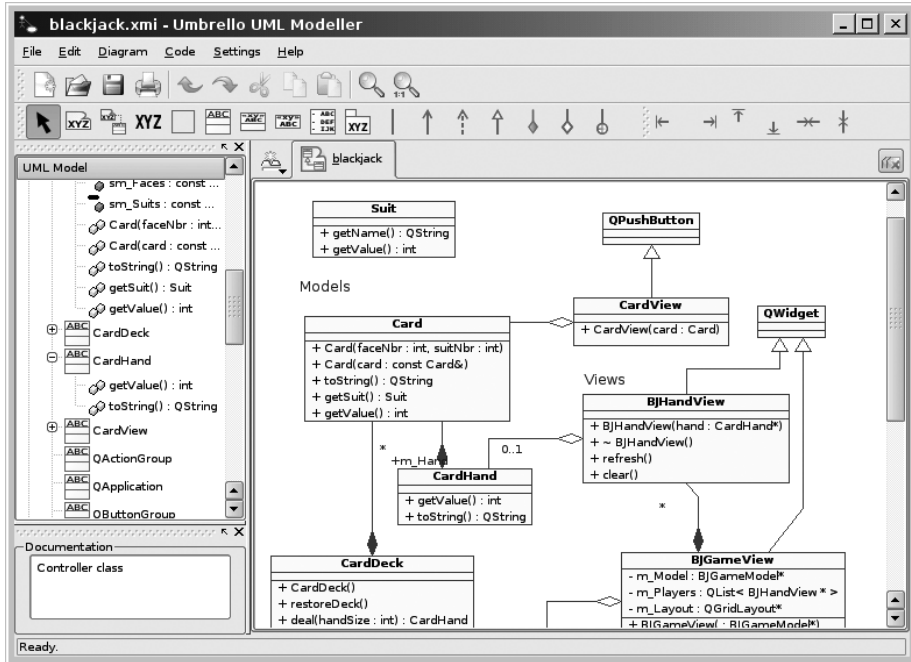


FIGURE C.3 Umbrello screenshot

Dia is a more general-purpose diagram tool with some UML features. There are many plugins and utilities that let you import code and export diagrams to and from Dia to other languages and formats.

¹⁶ <http://uml.sourceforge.net/index.php>

¹⁷ <http://www.gnome.org/projects/dia/>

C.5.2 jEdit

jEdit¹⁸ is a mature, open-source, programmer's text editor. Because it is written entirely in Java, it works on all platforms. Its keyboard configurability is very flexible—any action can be bound to a primary and an alternate shortcut.

To install it, first download a recent (5.0) version of the Java Development Kit (JDK) from <http://java.sun.com>, and then download the latest development version of jEdit.

Before using it very much, it is recommended you install some additional programs for development in C++:

- Plugins: Navigator, Project Viewer, Optional, FastOpen, Info Viewer, Console, Code Browser, XML
- Exuberant ctags version 5.5 or later (for use with Code Browser)
- ToggleHeaderSource version 0.4¹⁹ or later for (easy switching between header/source)

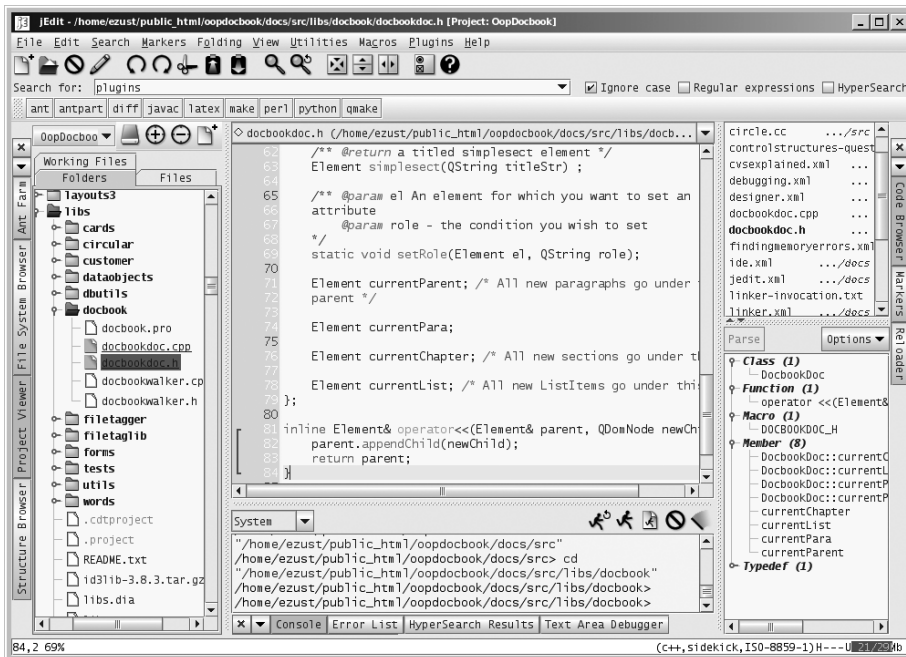


FIGURE C.4 jEdit screenshot

¹⁸ <http://www.jedit.org>

¹⁹ <http://community.jedit.org/?q=filestore/browse/34>



Check the keyboard configurability, Global Options -> Shortcuts. Notice that all the plugins and macros have their own shortcut-able actions. And after you have included some plugins, check out the Global Options -> Docking and dock some of the plugin's dockables to the sides of your edit window.

Bibliography

C++ References

[Josuttis99] *The C++ Standard Library*. Nicolai Josuttis. 1999. Addison-Wesley 0-201-37926-0.

[Meyers] *Effective C++*. Scott Meyers. 1999. Addison-Wesley. 0-201-56364-9.

[Stroustrup97] *The C++ Programming Language, Special Edition*. Bjarne Stroustrup. 2000. Addison-Wesley. 0-201-70073-5.

Qt References

[Blanchette06] *C++ GUI Programming with Qt 4*. Jasmin Blanchette and Mark Summerfield. 2006. Prentice Hall. 0-13-187249-4.

[Blanchette04] *C++ GUI Programming with Qt 3*. Jasmin Blanchette and Mark Summerfield. 2004. Prentice Hall. 0-13-124072-2.

[qtapistyle] *Designing Qt-Style C++ APIs*. Matthias Ettrich. 2005. Trolltech. <http://doc.trolltech.com/qq/qq13-apis.html>.

[qtestlib] *Writing Unittests for Qt 4 and KDE4 with QTestLib*. Brad Harris. 2005. developer.kde.org.

OOP References

[Buschmann96] *Pattern-Oriented Software Architecture*. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. John Wiley & Sons. 0-471-95869-7.

[Fowler04] *UML Distilled, Third Edition*. Martin Fowler. 2004. Addison-Wesley. 0-321-19368-7.

[Gamma95] *Design Patterns*. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Addison-Wesley. 0-201-63361-2.

[Martin98] *Pattern Languages of Program Design 3*. Robert C. Martin, Dirk Riehle, and Frank Buschmann. 1998. Addison-Wesley. pp. 293–312. 0-201-31011-2.

Docbook References

[docbook] *Docbook: The Definitive Guide*. Norman Walsh. 2005. O'Reilly Associates. <http://www.docbook.org/tdg/en/html/docbook.html>

[docbookxsl] *Docbook XSL: The Complete Guide*. Bob Stayton. 2005. SageHill Enterprises. <http://www.sagehill.net/docbookxsl/>

Miscellaneous References

[w3c] *w3c Recommendation: XHTML 1.0 The Extensible HyperText Markup Language*. 2005. W3C (World Wide Web Consortium). <http://www.w3.org/TR/xhtml1/>

[Friedl98] *Mastering Regular Expression, Second Edition*. Jeffrey Friedl. 1998. O'Reilly. 1-56592-257-3.

[Rehman03] *The Linux Development Platform*. Rafeeq Ur Rehman and Christopher Paul. 2003. Prentice Hall. 0-13-009115-4.



See Element Reference @ W3Schools.com (<http://www.w3schools.com/tags/default.asp>) for a good quick reference guide.

Index

Symbols and Numbers

- ansi switch, 13–14
- pedantic switch, 14
- Wall switch, 14
- (double dash), unary decrement operator, 26, 510
- #(pound character), for preprocessor directives, 15
- #define macro, 127
- #ifndef wrapper, 579
- #include
 - customizing using inheritance, 186–189
 - finding header files, 85–86
 - overview of, 15
 - preprocessor and, 579–581
 - unnecessary dependencies produced by, 174–175
- \$ (dollar sign), anchoring characters, 312
- % (percent sign), modulus operator, 26–27
- & (ampersand)
 - reference parameters using, 118
 - type modifier, 44
 - unary address-of operator, 36–38
- * (asterisk)
 - multiplication operator, 26
 - quantifier expressions, 311
 - unary deference operator, 37
- /* and */, in comments, 15
- *nix platform
 - fixing linker path, 177
 - installing libraries on, 176–178
 - open source development tools, 7–9
 - USER environment variable, 280
 - viewing manual pages, 14
- (...) (ellipsis), 542
- :/., pathname formats and, 250
- :: (file scope resolution operator), 468
- [] (square brackets), in command-line arguments, 158
- ^ (caret), anchoring characters, 312
- + (plus sign), addition operator, 25, 510
- ++ (double plus sign), unary increment operator, 26, 510
- < (less than), 27
- < > (angle brackets), #include directive, 85–86
- << (insertion operator), 16
- <= (less than or equal to), 27
- = (assignment operator). *See* Assignment operators
- != (not equal to), 27
- == (equal to), 27
- > (greater than), 27
- >= (greater than or equal to), 27
- ≧ (input operator), 16
- . (dot)
 - in bash shell, 178
 - member access and, 118
 - regular expression meta-character, 310
 - operator overloading and, 115
- ! (exclamation), unary not operator, 27
- ? (question mark), quantifier expressions, 311
- / (slash)
 - division operator, 26
 - as namespace delimiter, 567
- (subtraction operator), 25, 510
- , (comma operator), 115
- () parentheses
 - grouping and capturing characters, 312
 - macros and, 128
- 8-bit integer, 445–446

A

Abstract base classes, 148–152

Abstract Factory pattern

- benefits of, 369–372
- creating rules and friend functions, 366–369
- defined, 360
- exercises, 372–373
- importing objects, 376–380
- and libraries, 363–365
- overview of, 361–363
- qApp and Singleton pattern, 365–366

Abstract interfaces, multiple inheritance with, 531–532

Accessibility, 52

actionEvent(), 265–266

Adaptor pattern, 386–389

addAction(), 262, 264

Addition (+) operator, 25

AddLayout, widgets, 251–252

Address of (&), unary operators, 36–38

Addresses, pass-by-reference, 119

addSpacing(), 254–255

addStretch(), 254–255

addStrut(), 254–255

addWidget(), 251

Aggregate containers, 222–224

Aggregate relationships

- defined, 100
- pointer containers and, 221–224

Algorithms, generic, 225–227

Aliases, namespace, 475

amaroK, 427–428

American National Standards Institute.

- See* ANSI (American National Standards Institute)

Amperсанд character (&), reference

- parameters using, 44, 118

Amperсанд character (&), unary operator, 36–38

Anchoring characters, regular expressions, 312

Animation, QThread, 290–294

Anonymous namespaces, 476

ANSI (American National Standards Institute)

- ANSI C89, 575
- ANSI/ISO Draft Standard for C++, 6–7
- new operator, 515
- typecasts, 346, 450

Anti-patterns, 342–343

API (Application Programmer's Interface), 179

Applications, reusable components, 171

ArgumentList, 159–163

Arguments, processing command-line, 158–163

Arithmetic operators

- addition, 25, 510
- division, 26
- modulus, 26–27
- multiplication, 26
- overview of, 24–25
- pointers and, 510–511, 513–514
- subtraction, 25
- symbols for, 438

Array elements, 509

Arrays

- functions and return values and, 511–512
- kinds of, 513
- new failures and, 515–519
- overview of, 509–510
- reasons to avoid using in C++, 96
- review questions, 521
- summary, 519–520

Assignable data types, 221

Assignment operators

- with auto_ptr, 385
- copy, 64–67, 156
- for implicitly shared classes, 224
- pointers and, 513
- symbols for, 438

Assistant, Qt, 593–594

Associations, 101

Attributes

- Qt naming conventions, 90
- XML tags, 324

auto_ptr, 384–385, 388–389

B

Base classes, 136–140

- derivation from abstract, 148–152
- extending, 140–142
- inheritance and, 136
- initializing, 531
- member initialization for, 140
- order of initialization, 156
- overloading, function hiding, and overriding, 154–155

- parent objects *vs.*, 193
 - subclasses derived from, 137
 - virtual, 535–536
 - Bash scripts, 178
 - Behavioral patterns, 182
 - Bi-directional association, in QObject, 193
 - Binary operators, 115
 - Binding
 - compile-time, 144
 - run-time, 142, 144
 - Bitwise operators, 438
 - Block scope
 - vs.* file scope, 468–469
 - identifier, 466
 - overview of, 52–53, 465
 - statics defined inside, 63
 - Block statements, 480
 - Boolean
 - expressions, 480–481
 - operators, 27–28
 - types, 22–24
 - boost, 179
 - break
 - from loops, 484
 - from switch, 481
 - Button widgets, 239
 - byte, arrays, 513
- C**
- C
- C++ as extension of, 6
 - preprocessor, 579
 - standard Library, 578
 - “C with Objects,” 6
 - C++, 5–46
 - arithmetic operators, 25–29
 - brief history of, 6–7
 - const, 34–35, 40–43
 - first example, 12–16
 - identifiers, 19–22
 - input and output, 16–19
 - literals, 19–22
 - main() and command line
 - arguments, 24–25
 - overview of, 6
 - pointers and memory access, 36–40
 - preprocessor, 579
 - reference materials for, 601
 - reference variables, 43–44
 - reserved keywords, 575–576
 - scope options, 465–466
 - simple types, 22–24
 - standard Library, 578
 - standard library strings, 30–31
 - streams, 31–34
 - types, 19–22
 - variable initialization, 465
 - The C++ Programming Language* (Stroustrup), 6–7
 - Callbacks
 - defined, 327
 - importing objects with Abstract Factory, 380
 - case labels, 482
 - CaseIgnoreStrings, 227
 - Case-sensitivity, Qt naming conventions, 90
 - Casts, Casting. *See* Typecasting
 - catch statements
 - overview of, 490–494
 - rethrowing caught exceptions, 496
 - throw and, 502
 - Categories, QWidget, 239–240
 - Central widget, 270–272
 - cerr
 - global stream, 31–34
 - input and output, 16
 - char
 - arrays, 381, 513
 - throwing, 497
 - Character sets, regular expressions, 312
 - Character types, 22–24
 - characters(), 329
 - Children, QObject
 - Composite pattern, 196–199
 - environment variables, 281–282
 - finding, 199
 - management of, 194–196
 - overview of, 192–193
 - QProcess, 279
 - QWidget interacting with, 238
 - widget layout, 202–204, 252
 - Children, XML elements, 324
 - cin
 - global stream, 31–34
 - input and output, 16–19
 - cinclud2dot, 175
 - Circular dependencies, 582
 - class definitions
 - friend declarations within, 56
 - overview of, 49–51

- Class scope
 - defined, 51
 - identifier, 467
 - overview of, 466
- Class templates, generating generic containers, 216–219
- Classes, 47–79
 - const member functions, 68–78
 - constructors, 56–58
 - conversions, 67–68
 - copy constructors and assignment operators and, 64–67
 - definitions, 49–51, 464
 - destructors, 60
 - encapsulation, 54
 - form views, 400
 - friends of, 55–56
 - member access specifiers, 51–53
 - Qt naming conventions, 90
 - reusable components, 171
 - static keyword, 61–64
 - structs, 48–49
 - subobjects, 58–59
 - templates, 216–219
 - UML, 54–55
- className(), 344
- Client code, 51–53
- Code containers, 170–171
- Code reuse, 579
- CodeVisitor
 - customizing using inheritance, 186–189
 - decoupling, 188–189
- Comma operator (,), 115
- Command line arguments
 - main() function and, 24–25
 - processing, 158–163
- Command pattern, 262–267
- Comments, 15
- Comparison, pointer operations, 514
- Compilers
 - GNU C compiler (Gcc), 13–15
 - moc (Meta Object Compiler), 209–210
 - switches, 13–14
 - syntax errors, 587
- Compile-time
 - binding, 144
 - dependency, 173
- Complex numbers, 112–114
- Components
 - Composite pattern, 196–197
 - frameworks with reusable, 179
 - library, 179
- Composite pattern
 - DOM as application of, 330
 - managed containers and, 221–224
 - overview of, 196–197
 - QTreeWidgetItem as implementation of, 417–418
- Composition relationships, UML
 - defined, 55, 99
 - pointer containers and, 221–224
- Compound statements, 480
- Concrete class, 148
- Concurrency, 277–305
 - QProcess. *See* QProcess
 - QThread. *See* QThread
- Conditional
 - statements, 481–482
 - expressions, 28
- Conflicts, resolving multiple inheritance conflicts, 532–534
- Connect to slots, 203–204, 292
- const
 - const* and *const, 40–43
 - declaring reference parameter to be, 121–122
 - and globals, 471–472
 - implicitly shared classes vs., 225
 - members, 68–78
 - overloading on const-ness, 124–126
 - overview of, 34–35
 - pointers, 40–43, 513
- const_cast, 450–453
- Constructors (ctor)
 - conversion, 67–68
 - copy constructors, 64–67
 - exceptions and, 488
 - inheritance and, 155–157
 - overview of, 56–58
 - polymorphism from, 370–372
- Container widgets, 240
- Containers
 - arrays and, 513
 - class templates generating, 216–219
 - code, 170–171
 - defined, 96, 219
 - exercises and review questions, 233–235

- generics and, 219–221
 - implicitly shared, 224–225
 - managed, 221–224
 - overview of, 96–97
 - property, 355–356
 - Qt, 504
 - Serializer pattern, 227–229
 - sorted map example, 229–232
 - Context menus, 261
 - continue, loops, 484
 - Control, inversion of, 325
 - Control structures
 - defined, 479
 - exception expressions, 497–501
 - exception handling, 486
 - exceptions, 485
 - iteration structures, 483–485
 - rethrown exceptions and, 496–497
 - review questions, 502
 - throw statements, 486–488
 - try and catch statements, 490–494
 - Controller classes
 - defined, 284, 393
 - GUI development, 240
 - MP3 player, 553
 - Controller code, 392, 394–395
 - Controlling actions, 404–405
 - Convenience functions, ID3Lib, 384
 - Conversions
 - expressions, 447–449
 - overview of, 67–68
 - Copy assignment operators,
 - 65–67, 156
 - Copy constructors
 - assignment operators and, 64–67
 - for implicitly shared classes, 224
 - never inherited, 156–157
 - not public in QObject, 192
 - Core module, Qt, 91
 - cout
 - global stream, 31–34
 - input and output, 16
 - .cpp extension, class definitions, 50
 - CPPLIBS
 - as environment variable, 280
 - reusing other libraries, 171–172
 - Creational patterns, 360–372
 - applying, 360–361
 - benefits of, 369–372
 - defined, 182
 - exercises, 372–373
 - libraries and, 363–365
 - overview of, 361–363
 - qApp and Singleton pattern, 365–366
 - review questions, 390
 - rules and friend functions, 366–369
 - Cross-language mechanism, 280–281
 - ctor. *See* Constructors (ctor)
 - CustomerFactory, Abstract Factories,
 - 363–365
 - Cycle, 175
 - Cygwin, 12
- ## D
- Data members, Qt naming conventions, 90
 - Data model, Mp3File, 553–555
 - Data types
 - assignable, 221
 - GUI development and, 240
 - literals of, 20
 - Database models
 - GUI development and, 240
 - Qt SQL, 429–432
 - Database view, MP3 player, 569–571
 - DataObject
 - encoding/decoding as XML, 373–375
 - form model, 405–409
 - overview of, 353–354
 - DataObjectReader, 377–380
 - DataObjectTableModel, 412–417
 - Debugging
 - building debuggable target, 588–589
 - GNU debugger, 589–590
 - with loggers, 296–297
 - memory errors, 591–593
 - overview of, 587–588
 - Declarations
 - applying, 475
 - definitions compared with, 465
 - names, 464–465
 - Decoupling, 188–189
 - Decrement (—), unary operators, 26
 - Default arguments, 109
 - Default constructors, 57–58
 - Default labels, 482
 - Deference (*), unary operators, 37
 - Definitions
 - class, 49–51, 56, 464
 - declarations compared with, 465
 - environment variables on *nix, 178

- Definitions (*continued*)
 - object, function, and class, 464–465
 - polymorphic types, 524
 - private, protected, and public members, 52
 - Serializer pattern, 227–229
 - tables in MySQL, 425–426
 - template definitions in header files, 217
 - undefined pointers, 508
 - undefined reference to [identifier], 586–587
- delegates 360, 395, 405, 406
- delete operator
 - applying to pointers, 506–507
 - heap objects and, 470
 - overview of, 39
- Dependencies
 - circular, 582
 - compile-time, 173
 - customizing using inheritance, 187–188
 - defined, 173
 - managing library, 173–175
- Derivation
 - from abstract base class, 148–152
 - from ArgumentList, 160–163
 - kinds of, 138
 - polymorphism and, 142–147
 - public, protected, and private, 536–538
 - simple, 136–140
- Derived classes
 - employing inheritance using, 137–138
 - order of initialization, 156
 - overloading, function hiding, and overriding, 154–155
- Deserialization, playlists, 560
- Design, inheritance, 152–153
- Design patterns, 182–190
 - Abstract Factory pattern, 360, 361–363
 - Adaptor pattern, 386–389
 - anti-patterns, 342–343
 - Behaviorial patterns, 182
 - Command pattern, 262–267
 - Composite pattern, 196–199, 330
 - Creational pattern, 189–190
 - Facade pattern. *See* Facade patterns
 - implementing frameworks with, 179
 - Interpreter pattern, 524
 - Iteration and Visitor pattern, customizing, 184–189
 - MetaObject pattern, 344–345
 - Model-View-Controller (MVC), 392–393
 - Monostate pattern, 242
 - Observer (publish-subscribe) pattern, 200
 - overview of, 182
 - Reflection pattern, 344
 - Serializer pattern, 227–229, 373–380
 - Strategy pattern, 396
 - Visitor pattern, 182–189, 331–334
 - Wrapper pattern, 386
- Designer, Qt, 593–594
- DESTDIR variable, 176, 249
- Destructors (dtor)
 - exceptions and, 488
 - never inherited, 158
 - overview of, 60
 - static keyword and, 61–64
 - virtual, 526–528
- DevC++, 595
- Devel package, reusable components, 171
- Development environment, 579–599
 - building debuggable target, 588–589
 - debugging, 587–588
 - GNU debugger and, 588–590
 - jEdit, 598–599
 - linker, 582–584
 - linker error messages, 584–587
 - open source IDEs and development tools, 594–597
 - preprocessor, 579–581
 - Qt assistant and designer, 593–594
 - UML modeling tools, 597
- Development tools, open source, 594–597
- Dia, UML modeling tools, 597
- Dialogs
 - exercise, 248
 - input dialogs and widgets, 246–247
 - overview of, 244–246
- Directives, preprocessor, 475
- Directories
 - installing libraries in, 176
 - visiting code for, 183
- Display widgets, 240
- distort(), 300
- Division (/) operator, 26
- .dll file, 176
- do
 - loop, 484
- Docbook, 323, 602
- DocbookDoc class, 335–339

- DockWindows, 270–272
 - DOM (Document Object Model)
 - classes, 330
 - defined, 329
 - SAX *vs.*, 330
 - DomWalker, 332
 - Dot (.)
 - in bash shell, 178
 - operator overloading and, 115
 - do . . . while, iteration structures, 484
 - Downcasting. *See* RTTI
 - dtor. *See* Destructors (dtor)
 - Dynamic form models, 393–397
 - Dynamic memory, 511–512
 - Dynamic run-time binding, 144
 - dynamic_cast. *See also* Typecasting
 - defined, 345
 - qobject_cast similar to, 346
 - typecasting, 454–456
- E**
- Eclipse, 595
 - Editing, with macro expansion, 128
 - Editors, XML, 324
 - Elements, array, 509
 - Ellipsis (. . .), 542
 - else, conditional statement, 481
 - emit, 201, 205
 - Encapsulation, 54
 - Encryption, 130–132
 - endElement(), 378–379
 - endl, as manipulator, 17
 - Entries (array elements), 509
 - enum
 - converting strings to, 350
 - keyword, 443–445
 - Enumerations, 443–445
 - enumerator(), 350
 - Env command, 10
 - Environment variables
 - on *nix platform, 178
 - processes and, 280–281
 - Equivalence relation, 233–234
 - Errors
 - liability of macro expansion, 128–129
 - linker error messages, 584–587
 - Event loop, 201. *See also* QApplication, and event loop
 - Event-driven parsing, XML, 325–329
 - eventfilter(), Qonsole, 286–288
- Events
- Qonsole with keyboard, 286–288
 - QWidgets handling of, 238
- Exception
- expressions, 497–501
 - handling, 486
 - overview of, 485
 - rethrown, 496–497
 - safety, 302
 - throw() in function signature, 488–489
 - throw statements, 486–488
 - try statements, 490–494
- Explicit conversions (casts), 449
- explicit keyword, 68
- Exporting, to XML, 375–376
- Expressions
- evaluating logical expressions, 443
 - explicit conversions (typecasts), 449
 - standard conversions, 447–449
- Extended regular expressions,
 - Perl-style, 310–311
- Extending, 140–141
- eXtensible Markup Language. *See* XML (eXtensible Markup Language)
- extern keyword
 - declaring static objects, 476–477
 - file scope and, 467
 - global scope and, 466
- F**
- Façade patterns
- exercises, 389
 - Filetagger example, 385–389
 - functional, 384
 - overview of, 381–383
 - review questions, 390
 - smart pointers, 384–385
- Factories
- creating questions for forms with, 398–399
 - defined, 360
- Factory method, 360
- fifo (incoming message queue), 298
- File formats, MP3 player, 560
- File scope
 - vs.* block scope, 468–469
 - vs.* global scope, 466
 - overview of, 466
- Filenames, finding header files, 86
- Files, visiting code for, 183

- FileTagger
 - auto-generated form, 407
 - façade example, 386–389
 - MP3 player, 553, 568
 - SQL table, 426
 - FileVisitor
 - customizing using inheritance, 186–189
 - making into reusable tool, 184–186
 - Filters, MP3 player, 561–563
 - findChildren(), 199
 - Floating point numbers, 22–24
 - flush, as manipulator, 17
 - for loops, iteration structures, 484
 - Form views
 - dynamic form models, 395
 - for MP3 player, 568–569
 - overview of, 400–402
 - FormDialog, 400
 - FormFactory, 399
 - FormModel, 397–399, 405–409
 - Forms
 - defined, 393
 - dynamic model, 393–397
 - FormView, 395, 400–402
 - Forward declarations, 175, 580–582
 - Frameworks, library, 178–179
 - friend
 - keyword, 55–56
 - functions, 366–369
 - Functions, 105–133
 - declaring, 106–107
 - declaring inline, 126–127
 - defining, 464
 - ellipsis (. . .) and, 542
 - exceptions, 488–489
 - exercises and review questions, 130–133
 - global, 114
 - hiding, 154–155
 - inline vs. macro expansion, 127–130
 - invoking with QMetaObject, 344
 - main(), 24–25
 - operator overloading as, 111–116
 - with optional arguments, 109–111
 - overloading, 107–109, 154
 - overloading on const-ness, 124–126
 - overriding, 154
 - overview of, 105
 - passing parameters by reference, 118–121
 - passing parameters by value, 116–117
 - prototypes, 106–107
 - public, 54
 - QObjects can never be passed by value to any, 192
 - Qt naming conventions, 90
 - references to const, 121–122
 - return values, 122
 - returning references from, 122–124
 - scope, 465, 467
 - templates, 214–216
 - with variable-length argument lists, 542–543
 - virtual, 414
- ## G
- Garbage collection, 543
 - Gcc (GNU C compiler), 13–15
 - gdb (GNU debugger), 588–590
 - Generalization, 137
 - Generic containers, 96
 - Generics. *See also* Templates
 - algorithms and operators, 225–227
 - defined, 96
 - exercises and review questions, 233–235
 - templates, 214–219
 - getChar(), 279
 - getClassName(), 138–139
 - getline() function, 31
 - getSwitch(), 161
 - Global functions, 114
 - Global scope, 471
 - vs. file scope, 466
 - identifier, 466
 - partitioning into sub-scopes, 473
 - GNU C compiler (Gcc), 13–15
 - GNU debugger (gdb), 588–590
 - goto
 - avoiding in code, 468
 - switch statement and, 482
 - Graphic images, 248–251
 - Grouping characters, regular expressions, 312
 - Gui module, Qt, 91
- ## H
- handler, invoking parser, 325–326
 - Handler classes, 545
 - Header files
 - class definition defined in, 49–50
 - finding with #include, 85–86
 - libraries packaged as lib+, 170

- reusable components, 171
 - template definitions in, 217
 - Heap arrays, 96
 - Heap memory
 - benefits of factories, 369
 - corruption, 504
 - garbage collection and, 543
 - new operator allocating storage from, 38
 - pointer problems and, 506–508
 - storage class and, 470
 - Heavyweight objects, 355
 - Hiding functions, 154–155
 - Hierarchy, types, 22, 447
 - HOME, environment variable, 280
 - Host object, 68
 - HOSTNAME, environment variable, 280
 - HTML (HyperText Markup Language)
 - converting XML into, 335–336
 - uses of, 323
 - XML vs., 322–323
- I**
- ID3 tags, 381–383
 - reusing, 559–560
 - ID3Lib
 - convenience functions, 384
 - façade example, 385–389
 - overview of, 381–383
 - Identifiers
 - overview of, 19–22
 - scope of, 51, 465
 - Identity, QObject, 192–193
 - IDEs (integrated development environments)
 - finding header files within, 86
 - open source, 594–597
 - if statement, 481
 - Images, QWidget, 248–251
 - Implementation
 - class definitions, 50–51
 - of encapsulation, 54
 - relationships, 537
 - Implicitly shared containers, 224–225
 - Importing objects, with Abstract Factory, 376–380
 - Importing objects with Abstract Factory, SAX parser, 377
 - Include path, files, 85–86
 - Incomplete types, 581
 - Increment (++), unary operators, 26
 - Indexing, pointer operations, 514
 - indexOfProperty(), 350
 - Indirection
 - defined, 38
 - pointer operations, 514
 - Info command, 14
 - Inheritance, 135–165, 523–539. *See also* Multiple inheritance
 - base classes and, 136
 - client code example, 141–142
 - command-line arguments, processing, 158–160
 - constructors and, 155–157
 - copy assignment operators and, 156
 - copy constructors and, 156–157
 - defined, 136
 - derivation and ArgumentList, 160–163
 - derivation from abstract base class, 148–152
 - derivation with polymorphism, 142–147
 - design, 152–153
 - destructors and, 158
 - exercises and review questions, 163–165
 - function hiding, 154–155
 - member initialization and, 140, 531
 - multiple, 528–532
 - order of initialization, 156
 - overloading, 154
 - overriding, 154
 - polymorphism and virtual destructors, 526–528
 - public, protected, and private derivation, 536–538
 - QStringList and, 97–99
 - resolving multiple inheritance conflicts, 532–534
 - review questions, 539
 - simple derivation, 136–140
 - virtual base classes and, 535–536
 - virtual inheritance, 534–535
 - virtual pointers and virtual tables and, 524–526
 - visitor customization with, 186–189
 - inherits(), 347
 - Initialization
 - base class members, 140
 - class members, 531
 - static, 63–64
 - validators, 309

- Inline functions
 - #define macro *vs.*, 127
 - macro expansion *vs.*, 127–129
 - overview of, 126–127
 - Input and output, 16–19
 - Input dialogs
 - exercise, 248
 - and widgets, 246–247
 - Input widgets
 - defined, 239
 - dynamic form models, 396
 - form views, 402
 - overview of, 308–309
 - unforeseen types, 403–404
 - InputField
 - dynamic forms, 396–397
 - form views, 400–402
 - Insertion operator (\ll), 16
 - installEventFilter(), 288
 - instance()
 - AbstractFactory and, 361
 - Singleton pattern, 365
 - Instances, class definitions, 49
 - Instantiated, template, 215
 - int, Integer Types
 - arrays and, 512
 - enumerating, 443
 - overview of, 22–24
 - promotion, 447
 - signed and unsigned, 445–446
 - throwing, 497
 - Integrated development
 - environments (IDEs)
 - finding header files within, 86
 - open source, 594–597
 - Interface
 - generic containers, 96
 - relationships between classes, 536–537
 - Internationalization, QObject and, 211
 - Inversion of control, 325
 - iostream, 31
 - is-a relationships, 537
 - ISO, ANSI/ISO Draft Standard
 - for C++, 7
 - istream, 16–19
 - Item models, Qt 4, 409
 - Iteration
 - defined, 16
 - exercises, 101–103, 485
 - overview of, 97
 - QStringList and, 97–99
 - structures, 483–484
 - Iteration, and Visitor pattern, 182–190
 - customizing with inheritance, 186–189
 - exercises and review questions, 189–190
 - overview of, 184–186
 - QDir and QFileInfo (directories and files), 183
- J**
- JDBC classes, 429
 - JEdit, 598–599
 - join(), 97–99
- K**
- kdbg, 271
 - KDE 3.x (K Desktop Environment), 7
 - KDE debugger, 271
 - KDevelop, 595–596
 - Keyboard events, Qonsole with, 286–288
 - keyToValue(), 350
 - Keywords
 - C++ reserved, 575–576
 - const, 34–35
 - enum, 443–445
 - explicit, 68
 - extern, 467, 476–477
 - friend, 55–56
 - modifying simple types, 22
 - static, 61–64, 467
 - using, 475
 - virtual, 142–147
- L**
- Late run-time binding, 144
 - Layouts
 - GUI development, 240
 - QObject, 202–203
 - QWidgets. *See* QLayout, widgets
 - LD_LIBRARY_PATH, 176–177
 - Leaf nodes, Composite pattern, 197
 - lib files, 170
 - libcustomer, 363–365
 - libdataobjects, 363–365
 - libgtk++, 179
 - Libraries, 169–180
 - Abstract Factories and, 363–365
 - code containers, 170–171

- components, 179
 - defined, 169
 - dependency management, 173–175
 - finding header files within, 86
 - frameworks, 178–179
 - graphic image, 248–251
 - ID3Lib, 381–383
 - installing, 176–178, 585
 - overview of, 170
 - and plugins, 370
 - QWidget and, 239
 - reusing, 171–172
 - review questions, 180
 - LIBS variable, 172
 - libutils, 171
 - Linker
 - arguments to, 583
 - error messages, 584–587
 - linking process, 584
 - overview of, 582–584
 - path, 177
 - switches, 172
 - Link-time dependency, 173
 - The Linux Development Platform* (Rehman and Paul), 84
 - List view, media player, 552
 - Lists, 95–103
 - containers, 96–97
 - exercises and review questions, 101–103
 - iterators, 97–99
 - overview of, 95
 - relationships, 99–101
 - Literals, 19–22
 - Local variables, 350
 - Loggers
 - debugging with, 296–297
 - defined, 296
 - Logical expressions, evaluating, 443
 - Logical operators, 438
 - LogWindow, 296
 - loops
 - break and continue, 484
 - for, 484
 - lupdate tool, 211
 - Lvalue, 43
- M**
- M3U file format, 560
 - Macro expansion, 127–129
 - main()
 - overview of, 24–25
 - QObject child management, 194–196
 - QSettings, 243
 - make command
 - cleaning up files, 88–89
 - handling project files with, 84–85
 - overview of, 86–88
 - make dist command, 89
 - makedep dependency generator, 175
 - Makefile
 - cleaning up files, 89
 - example of qmake building, 86–88
 - overview of, 84
 - replaced in Qt by qmake, 85
 - man command, 14
 - Managed containers
 - implicitly shared, 224–225
 - overview of, 221–224
 - Manipulators
 - defined, 17
 - stream, 31–32
 - Manual pages, viewing on nix system, 14
 - Mapping layer, 415
 - Media players
 - components, 552–553
 - MP3 player view features, 563–564
 - Member access specifiers, 51–53
 - Member functions, 114
 - Member initialization, 57–58, 531
 - Member selection operators, 457–458
 - Memory access, 503–509
 - arrays and. *See* Arrays
 - overview of, 504
 - pointer problems and, 504–506
 - pointer problems with heap memory, 506–508
 - pointers, 36–40
 - review questions, 521
 - summary, 509
 - Memory allocation, thrashing and, 515
 - Memory corruption, 506
 - Memory heap. *See* Heap memory
 - Memory leaks, 506–507
 - Memory management operators, 438
 - Meta Object Compiler (moc), 209–210
 - Meta-characters, regular expression, 310–312
 - Metadata, MP3 songs, 381, 559
 - MetaObject pattern, 344–345, 373–375
 - methodCount(), 344

MinGW (Minimalist Gnu for Windows), 12

Mixed expressions, 27

moc (Meta Object Compiler), 209–210

modal attribute, 244

Models and views, 391–421

- controller code, 392
- controlling actions, 404–405
- DataObject form model, 405–409
- dynamic form models, 393–397
- form models, 397–399
- form views, 400–402
- GUI development, 240
- Model-View-Controller (MVC), 392–393
- Qt 4, 409–411
- review questions, 421
- separating models from views, 392
- table models, 411–417
- tree models, 417–420
- unforeseen types, 403–404

Model-View-Controller (MVC), 392–393

Modules, Qt 4, 91

Modulus (%) operator, 26–27

mono, 179

Monostate pattern, 242

Movie player

- QPixmap and animation, 290–294
- with QTimer, 294–295

MovieThread, QPixmap and animation, 290–294

MP3 files, 381, 553–555

MP3 jukebox assignments

- data model: Mp3File, 553–555
- database view, 569–571
- form view for FileTagger, 568–569
- ID3 tags, reusing, 559–560
- media player, 552–553
- MP3 player view features, 563–564
- persistent settings, 567–568
- play list models, 565
- play list serialization, 560
- Preference class, enumerating, 556–559
- queries and filters, 561–563
- source selector, 566–567
- testing Mp3File related classes, 561
- visitor generating playlists, 555–556

MSYS (from Minimalist Gnu for Windows), 12

Multiple inheritance

- with abstract interfaces, 531–532
- overview of, 528–529

QWidgets using, 238

resolving conflicts, 532–534

syntax, 529–531

Multiple threads, 296–302

Multiplication (*) operator, 26

Multithreaded environments, 369

MVC (Model-View-Controller), 392–393

MySQL, 424–427

- connecting from Qt, 425
- overview of, 424–425
- row insertion, 426–427
- table definition, 425–426

N

Namespaces

- aliases, 475
- anonymous, 476
- delimiter for, 567
- open, 476
- overview of, 15
- partitioning global scope into sub-scopes, 473
- reusable components, 171
- scope identifier, 467
- static objects and extern keyword and, 476–477
- using keyword and, 475

Naming conventions

- destructors, 60
- Qt guidelines, 90–91

Net module, Qt, 91

new operator

- failures, 515–519
- heap objects and, 470
- memory leaks and, 507
- overview of, 38–39

newObject(), 361–365, 380

nix platform. *See* *nix platform

Nodes, XML, 324, 330

Non-const reference parameters, 118–119

Not (!), unary operator, 27

nothrow, 544

NULL

- new failures and, 518–519
- pointers, 36, 506

O

Object files, 170

Object module, 171

- Object oriented programming (OOP), 601–602
 - ObjectFactory
 - Abstract Factories and libraries and, 362–365
 - managing singleton instance of, 365–366
 - in multithreaded applications, 369
 - Objects
 - changes to, 544–547
 - class definitions, 49, 464
 - defined, 36
 - global, 471
 - resource sharing and, 543
 - subobjects, 58–59
 - objectToXML (), 375–376
 - Observer (publish-subscribe) pattern, 200
 - Observer pattern, 200
 - OkAction, 404–405
 - One-to-many relationship, 99
 - One-to-one relationship, 99
 - Online resources
 - ANSI/ISO Draft Standard for C++, 7
 - downloading open source tarball, 9–11
 - gcc documentation, 14
 - qmake, 89
 - Qt, 89
 - Qt 4 Thread Support, 302
 - shell scripting, 178
 - OOP (object oriented programming), 601–602
 - Open namespaces, 476
 - Open source
 - defining, 7
 - downloading from source, 9–11
 - IDEs and development tools, 594–597
 - requiring Qt 4, 7–9
 - Operations, with pointers, 513–514
 - Operators, 438–442
 - arithmetic. *See* Arithmetic operators
 - assignment. *See* Assignment operators
 - binary, 115
 - boolean, 27–28
 - characteristics of, 439
 - classified by use, 438
 - delete, 39, 470, 506–507
 - generic, 225–227
 - insertion, 16
 - list of C++ operators, 440–442
 - member selection, 457–458
 - modulus, 26–27
 - new, 38–39, 470, 507, 515–519
 - overloading, 111–116
 - Run-Time Type Identification, 345–347
 - scope resolution, 50, 468
 - Serializer pattern and overloaded i/o, 227–229
 - shortcut, 26
 - sizeof (), 23–24
 - typecast, 346
 - typeid, 345
 - unary, 26–27, 36–38, 115
 - Optional arguments
 - enclosing in square brackets, 158
 - functions with, 109–111
 - Ostream, input and output, 16–19
 - Output. *See* Input and output
 - Overloading
 - on const-ness, 124–126
 - functions, 107–109, 154
 - operators, 111–116
 - unary operators and, 115
 - Overriding functions, 154
- ## P
- Parameters
 - command-line arguments, 158
 - function prototypes using, 106–107
 - optional arguments and, 109–111
 - QSettings string, 242
 - reference, declaring to be const, 121–122
 - reference, overview of, 118–121
 - template vs. function, 214
 - value, 116–117
 - Parents, QObject
 - base classes vs., 193
 - Composite pattern, 196–199
 - layout of widgets, 251
 - overview of, 192–193
 - QProcess, 279
 - QWidgets interacting with, 238
 - Parents, XML elements, 324
 - parse (), 325–326, 329
 - Parse event handler, 327
 - Parsers
 - event-driven, 325–329
 - SAX, 330–334, 377
 - XML, 327
 - Partitioning, global scope into sub-scopes, 473

- Pass-by-pointer, 120–121
 - Pass-by-reference, 120–121
 - Passive interface, 201, 326–327
 - PATH
 - as environment variable, 280
 - fixing linker path in Windows, 176–177
 - Paths, finding header files, 85–86
 - Patterns. *See* Design patterns
 - Performance, inline functions and, 126–127
 - Perl, regular expressions, 311
 - Persistent settings, MP3 player, 567–568
 - Play lists, MP3 player, 555–556, 561
 - Player view, MP3 player, 552, 563–564
 - Plug-ins
 - and libraries, 370
 - parsing XML with, 327
 - Pointers
 - arithmetic operators and, 510–511
 - const, 513
 - containers, 221
 - heap memory problems and, 506–508
 - memory access and, 36–40
 - operations with, 513–514
 - overview of, 22–24
 - problems due to improper handling of, 504–506
 - to QObject children, 192
 - smart, 39, 457
 - symbols for, 438
 - Polymorphism
 - from constructors, 370–372
 - defining polymorphic type, 524
 - derivation with, 142–147
 - exercises and review questions, 163–165
 - virtual destructors and, 526–528
 - POSIX (Portable Operating System Interface for UNIX), 7–9
 - Preference class, enumerating for MP3 player, 556–559
 - Prepared statements, 427
 - Preprocessor
 - development environment, 579–581
 - directives, 15
 - macros, 35
 - Primitives, 350
 - private derivation, 530, 536–538
 - private member, 52, 55
 - .pro file, 89
 - process(), 300
 - Process control. *See* QProcess
 - processDir(), 186
 - processFile(), 184–186, 187
 - Profiler, finding memory errors, 591–593
 - Program stack, storage class and, 470
 - Programming style, Qt guidelines, 90–91
 - Project files
 - cleaning up, 88–89
 - defined, 83
 - finding header files, 85–86
 - handling with make command, 83–85, 86–88
 - Promotion, expression conversion, 447
 - Properties
 - accessing, 350–352
 - containers (PropsMap), 355–356
 - describing QObject, 347–350
 - property(), 352
 - PropQuestion, 406, 408
 - PropsMap, 355–356, 362
 - protected derivation, 530, 536–538
 - protected member, 52, 137
 - public derivation, 530, 536–538
 - public functions, 54
 - Public interface, 54
 - public member, 52, 55
 - Pure virtual functions, in abstract base classes, 149–152
 - push(), 218
- ## Q
- Q_ENUM macro, 350
 - Q_PROPERTY macro, 347–350, 351
 - QAbstractItemModel, 417
 - QAbstractTableModel, 411–412, 414, 429
 - QAbstractxxxModel, 411
 - QActionGroups, 262–267
 - QActions
 - exercises, 267–269
 - implementing Command pattern, 262–267
 - QMenu, QMenuBar and, 260–262
 - Qtoolbars, QActionGroups and, 262–270
 - synchronizing data between model and view, 403–404
 - qApp
 - defined, 203
 - signals and slots, 204–209
 - Singleton pattern and, 365–366

- QApplication, and event loop, 200–209
 - connecting to slots, 203–204
 - layouts, 202–203
 - overview of, 200–202
 - signals and slots, 204–209
- QApplication, example creating, 82–83
- QBoxLayout, 251
- QByteArray, QSettings, 242–243
- QCache<Key,T>, 220
- QCoreApplication functions, 242
- QDate member functions, 92–93
- QDefaultxxxModel, 411
- qDeleteAll (), 222
- QDialog, 244–247
- QDir, 183
- QDockWidgets, 270–272
- QDomDocument, 330, 339
- QDomElement, 330, 333, 335–336
- QDomNode, 330–331, 333, 339
- QDoubleValidator, 308–309
- QEvents, 200–202, 286
- QFileInfo, 183
- QGridLayout, 251–260
- QHash<Key,T>, 220
- QHBoxLayout, 251
- QImage, 249
- QIntValidator, 308–309
- qjots application, 418–420
- QLabel, 82–83, 253
- QLayout, widgets, 251–260
 - exercises, 258–260
 - moving widgets across layouts, 256–258
 - overview of, 202–203, 251–254
 - spacing, stretching and struts, 254–255
- QLineEdit, 402
- QLinkedList<T>, 220
- QList, 96–97, 102
- QList<QString>, 220
- QList<T>, 219
- QListView, 417
- QMainWindow
 - managing dock window regions, 270–272
 - overview of, 240–241
 - QSettings and, 242–243
 - restoreState(), 243
 - saveState(), 242
- qmake
 - cleaning up files, 89
 - example of, 86–88
 - installing libraries, 176–177
 - online guide to, 89
 - overview of, 85
 - reusing other libraries, 172
 - downloading from source, 10
 - Win32 setup, 12
- QMap
 - example of, 229–234
 - implementing property containers, 355–356
- QMap<Key,T>, 220
- QMenu, 260–262, 267–270
- QMenuBar, 260–262, 267–270
- QMessageBox, 244–246
- QMetaObject, 344–345
- QMetaProperty
 - accessing properties, 352
 - describing QObject properties, 349–350
 - overview of, 344–345
- QModelIndex, 409, 411
- QMultiMap<Key,T>, 220
- QMutex, 302
- qobject_cast, 345–347
- QObject::inherits(), 345
- QObjectList, 192
- QObjects, 191–212
 - child management in, 194–196
 - Composite pattern, 196–199
 - connecting to slots, 203–204
 - DataObject extension of, 353–354
 - defined, 192
 - layouts, 202–203
 - moc and, 209–210
 - overview of, 192–193
 - QApplication and event loop, 200–209
 - QWidgets as, 238
 - review questions, 212
 - signals and slots, 204–209
 - storage class, 471
 - thread safety and, 302
 - tr() and internationalization, 211
 - values and objects, 210
- Qonsole
 - with keyboard events, 286–288
 - writing Xterm in Qt, 284–286
- QPaintDevice, 238
- QPicture, 249
- QPixmap
 - animation, 290–294
 - handling images, 249–251

- QProcess, 278–289
 - exercises, 288–289
 - overview of, 278–280
 - processes and environment, 280–283
 - Qconsole, 284–288
 - QThread *vs.*, 304
 - review questions, 305
- .qrc resource files, 248
- QRegExp
 - overview of, 310–312
 - phone number recognition, 313–316
 - regular expression validation, 316–317
- QRegExpValidator, 316–317
- QSemaphore, 302
- QSet<T>, 220
- QSettings, 242–243, 405
- QSlider widget, 293
- qSort(), 225–227
- QSplitter, 296
- QSqlQuery, 429
- QStack<T>, 220
- QStackedLayout, 251
- QStackedWidget, 251
- QString::arg(), 211
- QStringList
 - adding CaseIgnoreStrings to, 227
 - defined, 220
 - derivation and ArgumentList, 160–163
 - as implicitly shared classes, 225
 - input dialogs and widgets, 246–247
 - and iteration, 97–99, 101–102
 - processing command-line
 - arguments, 160
 - views of, 272–274
- QStrings
 - debugging and, 590
 - example using, 82–83
 - as implicitly shared classes, 225
 - input dialogs and widgets, 246–247
 - processing command-line arguments, 160
- Qt, 233–235
 - assistant and designer, 593–594
 - building with debugging symbols, 588
 - connecting to MySQL, 425
 - containers, 504
 - core modules, 91
 - dates, 91–93
 - defined, 81
 - exercises and review questions, 93–94
 - getting help online, 89
 - heap memory cleanup, 39
 - lists, 96
 - Makefile, 82–85
 - namespace delimiter, 567
 - project files, 83–89
 - QApplication and QLabel, 82–83
 - QDir and QFileInfo for visiting files, 183
 - qmake, 85
 - reference material, 601
 - setup, open source platforms, 7–11
 - setup, Win32, 12
 - streams, 91–93
 - style guides and naming conventions, 90
 - widgets. *See* QWidgets
 - XML Module, 325
- Qt 3, 7
- Qt 4
 - installing from packages, 8–9
 - models and views, 409–411
 - modules, 91
 - nix open source platform requiring, 7
 - reusable components of, 179
 - viewing version installed on your system, 8
- Qt Interest Mailing List, 89
- Qt source tarball, 10
- Qt SQL, 424–433
 - database models, 429–432
 - introduction to MySQL, 424–427
 - queries and result sets, 427–429
 - review questions, 433
- QTableView, 411–412, 414, 417,
429–431, 565, 569
- QtCentre, 89
- QTextStream, 31, 82–83, 91–93
- QThread, 290–304
 - exercises and review questions, 303–305
 - movie player with QTimer, 294–295
 - multiple threads, queues and loggers,
296–302
 - overview of, 290
 - QPixmap and animation, 290–294
 - QProcess *vs.*, 304
 - thread safety and QObjects, 302
 - using QTimer *vs.*, 295
- QTimer, 294–295
- QToolBar, 262–267
- QTreeView, 417
- QTreeWidgetItem, 417, 419
- Quantifiers, regular expressions, 311

- Queries
 - MP3 player, 561–563
 - Qt SQL, 427–429
 - Questions
 - dynamic form models, 395
 - form models, 397–399
 - rephrasing, 406
 - Queues, 296–302
 - QValidator, 308
 - QVariant, 350–352
 - QVBoxLayout, 251
 - QVector<T>, 220
 - QWaitCondition, 302
 - QWidget::addLayout, 251
 - QWidgets
 - categories, 239–240
 - defined, 238
 - dialogs, 244–248
 - images and resources, 248–251
 - layouts, 202–203, 251–260
 - overview of, 238–239
 - QActions, QMenuBar and QMenuBar, 260–262
 - QActions, Qttoolbars, and QActionGroups, 262–270
 - QMainWindow, 240–241
 - QSettings, 242–243
 - regions and QDockWidgets, 270–272
 - review questions, 275
 - sending QEvents, 201
 - signals and slots, 205–209
 - views of QStringList, 272–274
 - QXmlContentHandler, 326–327, 328
 - QXmlDefaultHandler, 328, 378
 - QXmlSimpleReader, 326–327
 - QXMLSimpleReader, 378
- R**
- rcc, resource compiler, 249
 - read(), 279
 - readAllStandardOutput(), 279
 - reader, invoking parser, 325–326
 - Reading strings, 33–34
 - readLine(), 279
 - readyReadStandardOutput(), 279
 - Refactoring, 136
 - Reference counting, 543
 - Reference parameters
 - declaring to be const, 121–122
 - overview of, 43–44, 118–121
 - Reference returns, from functions, 122–124
 - Reflective, defined, 343
 - Reflective programming, 341–358
 - anti-patterns, 342–343
 - DataObject, 353–355
 - exercises, 354–355
 - PropsMap, 355
 - Q_PROPERTY macro, 347–350
 - QMetaObject, 344–345
 - QVariant class, 350–352
 - review questions, 357
 - RTTI and qobject_cast, 345–347
 - Reflection pattern, 344
 - regex. *See* Regular expressions (regex)
 - Regions, QDockWidgets and, 270–272
 - Register, storage class, 470
 - Regular expressions (regex)
 - exercises and review questions, 318–319
 - overview of, 310–311
 - phone number recognition, 313–316
 - syntax, 311–312
 - validation, 316–317
 - reinterpret_cast, 453–454
 - Relational operators, 438
 - Relationships
 - defined, 55
 - exercises, 101
 - overview of, 99–101
 - review questions, 103
 - Reparenting, in QObject, 193
 - Required arguments, 158
 - Reserved keywords, C++, 575–576
 - Resource Collection File, 248
 - Resources
 - QWidgets, 248–251
 - sharing, 543, 547
 - restoreState(), QMainWindow, 243
 - Result sets, Qt SQL, 427–429
 - Rethrown, exceptions, 496–497
 - Return values
 - arrays, 511–512
 - functions, 122
 - Returning references, from functions, 122
 - Reusing, other libraries, 171–172
 - Root, of tree, 197
 - Row insertion, MySQL, 426–427
 - RTTI (run-time type identification), 454–458
 - dynamic_cast, 454–456
 - qobject_cast and, 345–347
 - typeid(), 456

- run(),
 - QProcess, 292
 - QThread, 300
- Run-time binding
 - dynamic or late, 144
 - enabling with virtual keyword, 142
- Run-time errors, debugging, 588
- run-time type identification. *See*
 - RTTI (run-time type identification)

S

- saveState(), QMainWindow, 242
- SAX parser
 - DOM *vs.*, 330
 - importing objects with Abstract Factory, 377
 - overview of, 331–334
- Scope
 - block, 52–53
 - class, 51
 - declaration determining, 464
 - file scope *vs.* block scope, 468–469
 - function, 465, 467
 - global, 471
 - identifier, 51, 465, 467
 - resolution operators, 50, 438
 - review questions, 478
 - storage class compared with, 470
 - types of, 465–468
- Searches, finding header files, 85–86
- Selection models, Qt 4, 409
- Selection statements
 - defined, 15
 - exercise, 483
 - overview of, 480–482
- Selector, media player, 552
- Serialization, playlist, 560
- Serializer pattern, 373–380
 - defining, 227–229
 - exporting to XML, 375–376
 - importing objects with Abstract Factory, 376–380
 - overview of, 373–375
 - review questions, 390
- setApplicationName(), 242
- setContent(), 330
- setGeometry() function, 250
- setOrganizationName(), 242
- setReadChannel(), 279

- Setup
 - open source platforms, 7–11
 - Win32, 12
- setupForm(), 309
- setValue()
 - form models, 397–398
 - QSettings, 242
- Sharing resources, 543, 547
- Shell scripting, 178
- Shortcut operators, 26
- Shout button, 202
- Signals
 - defined, 204
 - making concurrent code easier to read, 279
 - QMetaObject, 344
 - QObject, 204–209
 - slots and, 204–209
 - speaking with, 297–302
 - synchronous or asynchronous, 208
 - transmitting data to objects across threads, 292
- Signatures, function, 107
- Signed integral types, 445–446
- Simple statements, 480
- Simple types, 22–24, 29–30
- SimpleListApp, 272–274
- Singleton pattern
 - defined, 360–361
 - qApp and, 365–366
- Size, pointers, 24
- sizeof() operator, 23–24
- Slacker class, DOM tree walking, 332–334
- Slacker’s DocBook, 323
- Slash (/), as namespace delimiter, 567
- Slots
 - connecting signals with, 301–302
 - connections to, 203–204
 - defined, 204
 - making concurrent code easier to read, 279
 - QMetaObject, 344
 - signals and, 204–209
- Smart pointers, 39
 - auto_ptr, 384–385
 - member selection, 457
- Sorting, qSort(), 225–227
- Source code
 - libraries packaged as, 170
 - reusable components, 171

- Source selector, MP3 player, 553, 566–567
- Spacing, widget layout, 254–255
- Special characters, regular expressions, 310
- split(), QList and iteration, 97–99
- SQL, Qt, 424–433
 - database models, 429–432
 - introduction to MySQL, 424–427
 - overview of, 91
 - queries and result sets, 427–429
 - review questions, 433
- Standard headers, 577–578
- Standard Library (STL)
 - dynamic memory and, 504
 - finding header files within, 86
 - heap memory cleanup, 39
 - lists, 96
 - standard headers, 577–578
 - strings, 30–31
- Standards, C++, 6–7
- start()
 - processes, 278–279
 - threads, 291
- startElement(), 378–379
- State of the object, 49
- Statements
 - block, 480
 - compound, 480
 - conditional, 481–482
 - connect(), 292
 - defined, 479
 - overview of, 480
 - prepared, 427
 - review questions, 502
 - selection, 15, 480–483
 - simple, 480
 - switch, 481–482
 - throw, 486–488, 497–498
 - try and catch, 490–494
- static
 - binding, 144
 - block-scope, 63
 - keyword, 61–64, 467
 - local variables, 350
 - storage area, 470
 - using in Singleton pattern, 365
 - declaring, 476
 - namespaces and, 476–477
 - static_cast, 450
- stderr, 279
- std::list, 96
- stdout, 279
- STL (Standard Template Library). *See* Standard Library (STL)
- Storage class
 - const, 471–472
 - exercise, 472–473
 - globals, statics, and QObjects, 471
 - overview of, 470
 - register, 470
 - review questions, 478
- Strategy pattern, 396
- Streams, 31–34
- Stretching, widget layout, 254–255
- String literals, 20–21
- StringInputField, 402
- Strings
 - converting to enums, 350
 - QStringList and iteration, 97–99
 - reading, 33–34
 - Standard Library (STL), 30–31
 - writing, 32–33
- Stroustrup, Bjarne, 6–7
- struct
 - arrays of, 513
 - classes vs., 53
 - overview of, 48–49
- Structural patterns, 182
- Struts, widget layout, 254–255
- Subclasses
 - defined, 137
 - QLayout, 251
- SubObjects, 58–59
- Subtraction (–) operator, 25
- Suffolk University, 197–198
- superClass(), 344
- switch statement, 481–482
- Switched parameters, command-line arguments, 158
- Switches
 - command-line arguments, 158–163
 - compiler, 13–14
- Symbols, enclosing, 15
- Syntax
 - compiler errors, 587
 - multiple inheritance, 529–531
 - regular expressions (regex), 310–312
 - throw, 494

T

Table definition, MySQL, 425–426
 Table models, 411–416
 taglib, 381
 Tags, XML, 324
 Tarball

- downloading, 9
- overview of, 9

 TARGET, make command, 88
 target.path, make command, 88
 Template<class T>, 217
 Templates. *See also* Design patterns

- auto_ptr, 384–385
- causing generated code for each type, 350
- class, 216–219
- container, 219–221
- declaration code for, 217
- functions vs., 214–216
- generics and, 214–219
- instantiated, 215
- parameters, 214
- QList as template class, 96–97

 Text editors, 598
 this pointer, 68–70
 Thrashing, memory allocation and, 515
 Threads. *See* QThread
 Thread-safe objects, 302
 Throw(), in function signature, 488–489
 throw statements

- exception expressions and, 497–498
- overview of, 486–488
- uses of, 494

 Tilde (~) character, destructor

- names, 60

 toString(), 138–139, 142, 351–353
 tr(), and internationalization, 211
 Trailing arguments, functions with, 109
 Tree models, 417–420

- exercises, 420
- extended tree widget items, 418–420
- overview of, 417–418

 triggered() signals, QAction, 264
 Trolltech Online Documentation, 89
 try statements

- nesting, 498
- overview of, 490–494

 type() member function, 200
 Type modifier, 44
 Typecast operators, 346

Typecasting, 449–454

- ANSI standards, 450
- const_cast, 450–453
- C-style, 454
- downcasting, 333, 454
- dynamic_cast, 454–456
- overview of, 449
- reinterpret_cast, 453–454
- static_cast, 450
- typeid(), 456

 typeid operator, 345, 456–458, 524–525
 Type-restricted, 385
 Types, 437–461

- casting, 449–454
- conversion, 447
- enumeration, 443–445
- exercises, 458–460
- hierarchy of, 447
- logical expressions, 443
- member selection, 457–458
- operators, 438–442
- overview of, 19–22
- review questions, 461
- run-time type identification (RTTI), 454–456
- signed and unsigned integral types, 445–446
- simple, 22–24, 29–30
- variables, 49

U

Umbrello design tool, 54, 597
The Umbrello UML Modeller Handbook, 54
 UML (Unified Modeling Language)

- diagramming inheritance, 137
- inheritance design with, 153
- introduction to, 54–55
- modeling tools, 597
- relationships, 55

 Unable to Find libxxx.so.x, linker error messages, 585
 Unary operators

- address of (&), 36–38
- decrement (—), 26
- deference (*), 37
- increment (++), 26
- not (!), 27
- overloading and, 115

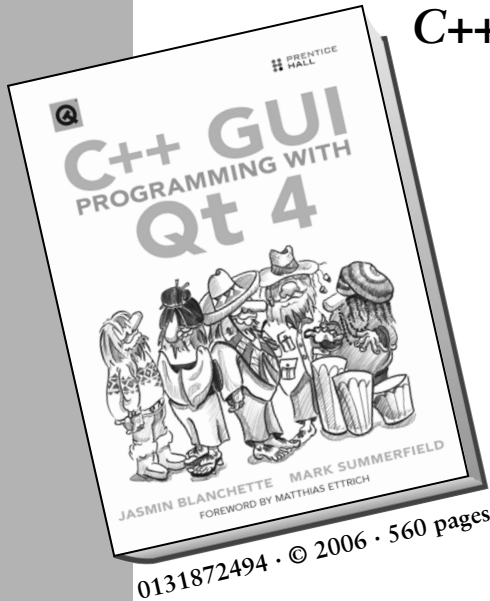
 Undefined pointers, 508

- Undefined Reference to [identifier], linker error messages, 586–587
 - Undefined Reference to vtable for
 - ClassName, linker error messages, 587
 - Unforeseen types, 403–404
 - Unified Modeling Language. *See* UML (Unified Modeling Language)
 - Union, 351
 - Unmanaged containers, 221–224
 - Unsigned integral types, 445–446
 - updateCursor(), 287–288
 - USER, as environment variable, 280
 - USERNAME, as environment variable, 280
 - using declaration, namespaces, 15, 475
 - /usr/local, 176
 - utils library, reusing, 171–172
- V**
- valgrind, profiler, 591–593
 - Validation, 307–319
 - exercises and review questions, 318–319
 - phone number recognition, 313–316
 - regular expressions, 316–317
 - regular expressions syntax, 310–312
 - using for, 310
 - validators, 308–309
 - Validators, 308–309
 - Value containers, 221
 - Value parameters, 116–117
 - Values
 - function return, 122
 - QObject, 210
 - Variable-length argument lists, 542–543
 - Variables
 - class types, 49
 - const, 471–472
 - declarations, 15
 - defined, 36
 - environment variables, 280–282
 - global, 466
 - initialization in C++, 465
 - local, 350
 - reference, 43–44
 - vector class, 488–489
 - Views. *See also* Models and views
 - form, 400–402
 - Qt 4, 409–411
 - separating models from views, 392
 - Virtual base classes, 535–536
 - Virtual destructors, 526–528
 - virtual functions
 - overriding and, 154
 - pure, 149–152
 - QAbstractTableModel, 414
 - Virtual inheritance, 534–535
 - virtual keyword
 - derivation with polymorphism using, 142–147
 - enabling runtime binding with, 142
 - virtual methods, parsing XML with, 327
 - Virtual pointers, 524–526
 - Virtual tables (vtables), inheritance and, 524–526
 - Visibility, 52
 - Visitor, generating playlists, 555–556
 - Visitor pattern, iteration and, 182–190
 - customizing with inheritance, 186–189
 - DOM tree walking, 331–334
 - exercises and review questions, 189–190
 - overview of, 184–186
 - QDir and QFileInfo (directories and files), 183
 - void, 22
 - Vtables (virtual tables), 524
- W**
- walkTree() method, 333
 - while, iteration structures, 483–484
 - Whitespace, pointer problems and, 505
 - Widgets. *See also* QWidgets
 - displaying current play list on MP3 player, 565
 - Qt designer, 594
 - tree, 417–420
 - Win32, setup, 12
 - Window, QWidget, 238
 - Windows
 - environment variables, 281
 - installing libraries in, 176–177
 - USERNAME environment variable, 280
 - Wrappers
 - FileTagger (façade example), 386–389
 - header files, 50
 - using auto_ptr in, 384–385
 - write(), 279
 - Writing strings, 32–33
 - Wt⁸, 179
 - wxWidgets, 179

X

- XML (eXtensible Markup Language),
 - 321–340
 - encoding/decoding DataObjects as,
 - 373–375
 - event-driven parsing, 325–329
 - exercises and review questions, 339
 - exporting to, 375–376
 - generating output with DOM,
 - 335–339
 - HTML vs., 321–323
 - importing objects with Abstract Factory,
 - 376–380
 - nodes, 324
 - Qt XML Module, 325
 - tree structures and DOM, 329–334
 - XML editors, 324
- Xml module, Qt, 91, 325
- xmllint, 325
- Xterm, 284–286

The Only Official Best-Practice Guide to Qt 4.1 Programming



C++ GUI Programming with Qt 4

**Jasmin Blanchette and
Mark Summerfield**

Using Trolltech's Qt you can build industrial-strength C++ applications that run natively on Windows, Linux/Unix, Mac OS X, and embedded Linux—without making source code changes. With *C++ GUI Programming with Qt 4*, Trolltech insiders have written a start-to-finish guide to getting great results with the most powerful version of Qt ever created: Qt 4.1.

Using this book, you'll discover the most effective Qt 4 programming patterns and techniques and master key technologies ranging from Qt's model/view architecture to Qt's powerful new 2D paint engine.

The authors provide you with unparalleled insight into Qt's event model and layout system. Then, using realistic examples, they introduce superior techniques for everything from basic GUI development to advanced database and XML integration.

- Includes new chapters on Qt 4's model/view architecture and Qt's new plugin support, along with a brief introduction to Qtopia embedded programming
- Covers all Qt fundamentals, from dialogs and windows to implementing application functionality
- Introduces best practices for layout management and event processing
- Shows how to make the most of Qt 4's new APIs, including the powerful new 2D paint engine and the new easy-to-use container classes
- Contains completely updated material in every chapter
- Presents advanced Qt 4 techniques covered in no other book, from creating both Qt and application plugins to interfacing with native APIs
- Contains an in-depth appendix on C++/Qt programming for experienced Java developers

The accompanying CD-ROM includes the open source edition of Qt 4.1.1 for Windows, Mac, Linux, and many Unixes, as well as MinGW, a set of freely available development tools that can be used to build Qt applications on Windows and was used to create the source code for the book's examples.



YOUR GUIDE TO IT REFERENCE



Articles

Keep your edge with thousands of free articles, in-depth features, interviews, and IT reference recommendations – all written by experts you know and trust.



Online Books

Answers in an instant from **InformIT Online Book's** 600+ fully searchable on line books. For a limited time, you can get your first 14 days **free**.

POWERED BY
Safari
TECH BOOKS ONLINE



Catalog

Review online sample chapters, author biographies and customer rankings and choose exactly the right book from a selection of over 5,000 titles.

http://www.phptr.com/

Prentice Hall PTR InformIT InformIT Online Books Financial Times Prentice Hall ft.com PTG Interactive Reuters



TOMORROW'S SOLUTIONS FOR TODAY'S PROFESSIONALS

Prentice Hall

Professional Technical Reference

Browse

Book Series

What's New

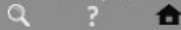
User Groups

Alliances

Special Sales

Contact Us

Search | Help | Home



Quick Search

PTR Favorites

Find a Bookstore

Book Series

Special Interests

Newsletters

Press Room

International

Best Sellers

Solutions Beyond
the Book

Shopping Bag

Keep Up to Date with

PH PTR Online

We strive to stay on the cutting edge of what's happening in professional computer science and engineering. Here's a bit of what you'll find when you stop by www.phptr.com:



What's new at PHPTR? We don't just publish books for the professional community, we're a part of it. Check out our convention schedule, keep up with your favorite authors, and get the latest reviews and press releases on topics of interest to you.



Special interest areas offering our latest books, book series, features of the month, related links, and other useful information to help you get the job done.



User Groups Prentice Hall Professional Technical Reference's User Group Program helps volunteer, not-for-profit user groups provide their members with training and information about cutting-edge technology.



Companion Websites Our Companion Websites provide valuable solutions beyond the book. Here you can download the source code, get updates and corrections, chat with other users and the author about the book, or discover links to other websites on this topic.



Need to find a bookstore? Chances are, there's a bookseller near you that carries a broad selection of PTR titles. Locate a Magnet bookstore near you at www.phptr.com.



Subscribe today! Join PHPTR's monthly email newsletter! Want to be kept up-to-date on your area of interest? Choose a targeted category on our website, and we'll keep you informed of the latest PHPTR products, author events, reviews and conferences in your interest area.

Visit our mailroom to subscribe today! http://www.phptr.com/mail_lists



THIS BOOK IS SAFARI ENABLED

INCLUDES FREE 45-DAY ACCESS TO THE ONLINE EDITION

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

TO GAIN 45-DAY SAFARI ENABLED ACCESS TO THIS BOOK:

- Go to <http://www.prenhallprofessional.com/safarienabled>
- Complete the brief registration form
- Enter the coupon code found in the front of this book on the "Copyright" page

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.



PRENTICE
HALL