Modern C++ Design

Generic Programming and Design Patterns Applied

Andrei Alexandrescu

Foreword by Scott Meyers

Foreword by John Vlissides

C++ In-Depth Series • Bjarne Stroustrup

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Modern C++ Design

The C++ In-Depth Series

Bjarne Stroustrup, Editor

"I have made this letter longer than usual, because I lack the time to make it short."
—BLAISE PASCAL

The advent of the ISO/ANSI C++ standard marked the beginning of a new era for C++ programmers. The standard offers many new facilities and opportunities, but how can a real-world programmer find the time to discover the key nuggets of wisdom within this mass of information? **The C++ In-Depth Series** minimizes learning time and confusion by giving programmers concise, focused guides to specific topics.

Each book in this series presents a single topic, at a technical level appropriate to that topic. The Series' practical approach is designed to lift professionals to their next level of programming skills. Written by experts in the field, these short, in-depth monographs can be read and referenced without the distraction of unrelated material. The books are cross-referenced within the Series, and also reference *The C++ Programming Language* by Bjarne Stroustrup.

As you develop your skills in C++, it becomes increasingly important to separate essential information from hype and glitz, and to find the in-depth content you need in order to grow. The C++ In-Depth Series provides the tools, concepts, techniques, and new approaches to C++ that will give you a critical edge.

Titles in the Series

Accelerated C++: Practical Programming by Example, Andrew Koenig and Barbara E. Moo

Applied C++: Practical Techniques for Building Better Software, Philip Romanik and Amy Muntz

The Boost Graph Library: User Guide and Reference Manual, Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Herb Sutter and Andrei Alexandrescu

C++ In-Depth Box Set, Bjarne Stroustrup, Andrei Alexandrescu, Andrew Koenig, Barbara E. Moo, Stanley B. Lippman, and Herb Sutter

C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns, Douglas C. Schmidt and Stephen D. Huston

C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks, Douglas C. Schmidt and Stephen D. Huston

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, David Abrahams and Aleksey Gurtovoy

Essential C++, Stanley B. Lippman

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu

More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

For more information, check out the series web site at www.awprofessional.com/series/indepth/

Modern C++ Design

Generic Programming
and Design Patterns Applied

Andrei Alexandrescu

Addison-Wesley

Boston • 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 Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author 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 discounts on this book when ordered in quantity for special sales. For more information, please contact:

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
Alexandrescu, Andrei.
Modern C++ design : generic programming and design patterns applied / Andrei Alexandrescu.
p. cm. — (C++ in depth series)
Includes bibliographical references and index.
ISBN 0-201-70431-5
1. C++ (Computer program language) 2. Generic programming (Computer science) I. Title. II. Series.
QA76.73.C153 A42 2001
005.13′3—dc21 00-049596

Copyright © 2001 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ISBN 0-201-70431-5
Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.
19th Printing January 2011

Contents

Foreword by Scott Meyers		**xi**
Foreword by John Vlissides		**xv**
Preface		**xvii**
Acknowledgments		**xxi**
Part I	**Techniques**	**1**
Chapter 1	**Policy-Based Class Design**	**3**
1.1	The Multiplicity of Software Design	3
1.2	The Failure of the Do-It-All Interface	4
1.3	Multiple Inheritance to the Rescue?	5
1.4	The Benefit of Templates	6
1.5	Policies and Policy Classes	7
1.6	Enriched Policies	12
1.7	Destructors of Policy Classes	12
1.8	Optional Functionality Through Incomplete Instantiation	13
1.9	Combining Policy Classes	14
1.10	Customizing Structure with Policy Classes	16
1.11	Compatible and Incompatible Policies	17
1.12	Decomposing a Class into Policies	19
1.13	Summary	20

Chapter 2	**Techniques**	**23**
2.1	Compile-Time Assertions	23
2.2	Partial Template Specialization	26
2.3	Local Classes	28
2.4	Mapping Integral Constants to Types	29
2.5	Type-to-Type Mapping	31
2.6	Type Selection	33
2.7	Detecting Convertibility and Inheritance at Compile Time	34
2.8	A Wrapper Around `type_info`	37
2.9	`NullType` and `EmptyType`	39
2.10	Type Traits	40
2.11	Summary	46
Chapter 3	**Typelists**	**49**
3.1	The Need for Typelists	49
3.2	Defining Typelists	51
3.3	Linearizing Typelist Creation	52
3.4	Calculating Length	53
3.5	Intermezzo	54
3.6	Indexed Access	55
3.7	Searching Typelists	56
3.8	Appending to Typelists	57
3.9	Erasing a Type from a Typelist	58
3.10	Erasing Duplicates	59
3.11	Replacing an Element in a Typelist	60
3.12	Partially Ordering Typelists	61
3.13	Class Generation with Typelists	64
3.14	Summary	74
3.15	`Typelist` Quick Facts	75
Chapter 4	**Small-Object Allocation**	**77**
4.1	The Default Free Store Allocator	78
4.2	The Workings of a Memory Allocator	78
4.3	A Small-Object Allocator	80
4.4	`Chunks`	81
4.5	The Fixed-Size Allocator	84
4.6	The `SmallObjAllocator` Class	87
4.7	A Hat Trick	89
4.8	Simple, Complicated, Yet Simple in the End	92
4.9	Administrivia	93
4.10	Summary	94
4.11	Small-Object Allocator Quick Facts	94

Part II Components 97

Chapter 5 Generalized Functors 99
5.1 The Command Design Pattern 100
5.2 Command in the Real World 102
5.3 C++ Callable Entities 103
5.4 The `Functor` Class Template Skeleton 104
5.5 Implementing the Forwarding `Functor::operator()` 108
5.6 Handling Functors 110
5.7 Build One, Get One Free 112
5.8 Argument and Return Type Conversions 114
5.9 Handling Pointers to Member Functions 115
5.10 Binding 119
5.11 Chaining Requests 122
5.12 Real-World Issues I: The Cost of Forwarding Functions 122
5.13 Real-World Issues II: Heap Allocation 124
5.14 Implementing Undo and Redo with `Functor` 125
5.15 Summary 126
5.16 `Functor` Quick Facts 126

Chapter 6 Implementing Singletons 129
6.1 Static Data + Static Functions != Singleton 130
6.2 The Basic C++ Idioms Supporting Singleton 131
6.3 Enforcing the Singleton's Uniqueness 132
6.4 Destroying the Singleton 133
6.5 The Dead Reference Problem 135
6.6 Addressing the Dead Reference Problem (I): The Phoenix Singleton 137
6.7 Addressing the Dead Reference Problem (II): Singletons with Longevity 139
6.8 Implementing Singletons with Longevity 142
6.9 Living in a Multithreaded World 145
6.10 Putting It All Together 148
6.11 Working with `SingletonHolder` 153
6.12 Summary 155
6.13 `SingletonHolder` Class Template Quick Facts 155

Chapter 7 Smart Pointers 157
7.1 Smart Pointers 101 157
7.2 The Deal 158
7.3 Storage of Smart Pointers 160
7.4 Smart Pointer Member Functions 161

7.5	Ownership-Handling Strategies	163
7.6	The Address-of Operator	170
7.7	Implicit Conversion to Raw Pointer Types	171
7.8	Equality and Inequality	173
7.9	Ordering Comparisons	178
7.10	Checking and Error Reporting	181
7.11	Smart Pointers to `const` and `const` Smart Pointers	182
7.12	Arrays	183
7.13	Smart Pointers and Multithreading	184
7.14	Putting It All Together	187
7.15	Summary	194
7.16	`SmartPtr` Quick Facts	194
Chapter 8	**Object Factories**	**197**
8.1	The Need for Object Factories	198
8.2	Object Factories in C++: Classes and Objects	200
8.3	Implementing an Object Factory	201
8.4	Type Identifiers	206
8.5	Generalization	207
8.6	Minutiae	210
8.7	Clone Factories	211
8.8	Using Object Factories with Other Generic Components	215
8.9	Summary	216
8.10	`Factory` Class Template Quick Facts	216
8.11	`CloneFactory` Class Template Quick Facts	217
Chapter 9	**Abstract Factory**	**219**
9.1	The Architectural Role of Abstract Factory	219
9.2	A Generic Abstract Factory Interface	223
9.3	Implementing `AbstractFactory`	226
9.4	A Prototype-Based Abstract Factory Implementation	228
9.5	Summary	233
9.6	`AbstractFactory` and `ConcreteFactory` Quick Facts	233
Chapter 10	**Visitor**	**235**
10.1	Visitor Basics	235
10.2	Overloading and the Catch-All Function	242
10.3	An Implementation Refinement: The Acyclic Visitor	243
10.4	A Generic Implementation of Visitor	248
10.5	Back to the "Cyclic" Visitor	255
10.6	Hooking Variations	258
10.7	Summary	260
10.8	`Visitor` Generic Component Quick Facts	261

Chapter 11	**Multimethods**	**263**
11.1	What Are Multimethods?	264
11.2	When Are Multimethods Needed?	264
11.3	Double Switch-on-Type: Brute Force	265
11.4	The Brute-Force Approach Automated	268
11.5	Symmetry with the Brute-Force Dispatcher	273
11.6	The Logarithmic Double Dispatcher	276
11.7	`FnDispatcher` and Symmetry	282
11.8	Double Dispatch to Functors	282
11.9	Converting Arguments: `static_cast` or `dynamic_cast`?	285
11.10	Constant-Time Multimethods: Raw Speed	290
11.11	`BasicDispatcher` and `BasicFastDispatcher` as Policies	293
11.12	Looking Forward	294
11.13	Summary	296
11.14	Double Dispatcher Quick Facts	297
Appendix	**A Minimalist Multithreading Library**	**301**
A.1	A Critique of Multithreading	302
A.2	Loki's Approach	303
A.3	Atomic Operations on Integral Types	303
A.4	Mutexes	305
A.5	Locking Semantics in Object-Oriented Programming	306
A.6	Optional `volatile` Modifier	308
A.7	Semaphores, Events, and Other Good Things	309
A.8	Summary	309
Bibliography		**311**
Index		**313**

This page intentionally left blank

Foreword

by Scott Meyers

In 1991, I wrote the first edition of *Effective* C++. The book contained almost no discussions of templates, because templates were such a recent addition to the language, I knew almost nothing about them. What little template code I included, I had verified by e-mailing it to other people, because none of the compilers to which I had access offered support for templates.

In 1995, I wrote *More Effective* C++. Again I wrote almost nothing about templates. What stopped me this time was neither a lack of knowledge of templates (my initial outline for the book included an entire chapter on the topic) nor shortcomings on the part of my compilers. Instead, it was a suspicion that the C++ community's understanding of templates was about to undergo such dramatic change, anything I had to say about them would soon be considered trite, superficial, or just plain wrong.

There were two reasons for that suspicion. The first was a column by John Barton and Lee Nackman in the January 1995 C++ *Report* that described how templates could be used to perform typesafe dimensional analysis with zero runtime cost. This was a problem I'd spent some time on myself, and I knew that many had searched for a solution, but none had succeeded. Barton and Nackman's revolutionary approach made me realize that templates were good for a lot more than just creating containers of `T`.

As an example of their design, consider this code for multiplying two physical quantities of arbitrary dimensional type:

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                        Physical<m2, l2, t2> rhs)
{
   return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

Even without the context of the column to clarify this code, it's clear that this function template takes six parameters, none of which represents a type! This use of templates was such a revelation to me, I was positively giddy.

Shortly thereafter, I started reading about the STL. Alexander Stepanov's elegant library design, where containers know nothing about algorithms; algorithms know nothing about

containers; iterators act like pointers (but may be objects instead); containers and algorithms accept function pointers and function objects with equal aplomb; and library clients may extend the library without having to inherit from any base classes or redefine any virtual functions, made me feel—as I had when I read Barton and Nackman's work—like I knew almost *nothing* about templates.

So I wrote almost nothing about them in *More Effective* C++. How could I? My understanding of templates was still at the containers-of-`T` stage, while Barton, Nackman, Stepanov, and others were demonstrating that such uses barely scratched the surface of what templates could do.

In 1998, Andrei Alexandrescu and I began an e-mail correspondence, and it was not long before I recognized that I was again about to modify my thinking about templates. Where Barton, Nackman, and Stepanov had stunned me with what templates could *do,* however, Andrei's work initially made more of an impression on me for *how* it did what it did.

One of the simplest things he helped popularize continues to be the example I use when introducing people to his work. It's the `CTAssert` template, analogous in use to the `assert` macro, but applied to conditions that can be evaluated during compilation. Here it is:

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

That's it. Notice how the general template, `CTAssert`, is never defined. Notice how there is a specialization for `true`, but not for `false`. In this design, what's *missing* is at least as important as what's present. It makes you look at template code in a new way, because large portions of the "source code" are deliberately omitted. That's a very different way of thinking from the one most of us are used to. (In this book, Andrei discusses the more sophisticated `CompileTimeChecker` template instead of `CTAssert`.)

Eventually, Andrei turned his attention to the development of template-based implementations of popular language idioms and design patterns, especially the GoF* patterns. This led to a brief skirmish with the Patterns community, because one of their fundamental tenets is that patterns cannot be represented in code. Once it became clear that Andrei was automating the generation of pattern *implementations* rather than trying to encode patterns themselves, that objection was removed, and I was pleased to see Andrei and one of the GoF (John Vlissides) collaborate on two columns in the C++ *Report* focusing on Andrei's work.

In the course of developing the templates to generate idiom and pattern implementations, Andrei was forced to confront the variety of design decisions that all implementers face. Should the code be thread safe? Should auxiliary memory come from the heap, from the stack, or from a static pool? Should smart pointers be checked for nullness prior to dereferencing? What should happen during program shutdown if one Singleton's destructor tries to use another Singleton that's already been destroyed? Andrei's goal was to offer his clients all possible design choices while mandating none.

*"GoF" stands for "Gang of Four" and refers to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the definitive book on patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

His solution was to encapsulate such decisions in the form of *policy classes,* to allow clients to pass policy classes as template parameters, and to provide reasonable default values for such classes so that most clients could ignore them. The results can be astonishing. For example, the Smart Pointer template in this book takes only 4 policy parameters, but it can generate over 300 different smart pointer types, each with unique behavioral characteristics! Programmers who are content with the default smart pointer behavior, however, can ignore the policy parameters, specify only the type of object pointed to by the smart pointer, and reap the benefits of a finely crafted smart pointer class with virtually no effort.

In the end, this book tells three different technical stories, each compelling in its own way. First, it offers new insights into the power and flexibility of C++ templates. (If the material on typelists doesn't knock your socks off, it's got to be because you're already barefoot.) Second, it identifies orthogonal dimensions along which idiom and pattern implementations may differ. This is critical information for template designers and pattern implementers, but you're unlikely to find this kind of analysis in most idiom or pattern descriptions. Finally, the source code to Loki (the template library described in this book) is available for free download, so you can study Andrei's implementation of the templates corresponding to the idioms and patterns he discusses. Aside from providing a nice stress test for your compilers' support for templates, this source code serves as an invaluable starting point for templates of your own design. Of course, it's also perfectly respectable (and completely legal) to use Andrei's code right out of the box. I know he'd want you to take advantage of his efforts.

From what I can tell, the template landscape is changing almost as quickly now as it was in 1995 when I decided to avoid writing about it. At the rate things continue to develop, I may *never* write about templates. Fortunately for all of us, some people are braver than I am. Andrei is one such pioneer. I think you'll get a lot out of his book. I did.

Scott Meyers
September 2000

This page intentionally left blank

Foreword

by John Vlissides

What's left to say about C++ that hasn't already been said? Plenty, it turns out. This book documents a convergence of programming techniques—generic programming, template metaprogramming, object-oriented programming, and design patterns—that are well understood in isolation but whose synergies are only beginning to be appreciated. These synergies have opened up whole new vistas for C++, not just for programming but for software design itself, with profound implications for software analysis and architecture as well.

Andrei's generic components raise the level of abstraction high enough to make C++ begin to look and feel like a design specification language. Unlike dedicated design languages, however, you retain the full expressiveness and familiarity of C++. Andrei shows you how to program in terms of design concepts: singletons, visitors, proxies, abstract factories, and more. You can even vary implementation trade-offs through template parameters, with positively no runtime overhead. And you don't have to blow big bucks on new development tools or learn reams of methodological mumbo jumbo. All you need is a trusty, late-model C++ compiler—and this book.

Code generators have held comparable promise for years, but my own research and practical experience have convinced me that, in the end, code generation doesn't compare. You have the round-trip problem, the not-enough-code-worth-generating problem, the inflexible-generator problem, the inscrutable-generated-code problem, and of course the I-can't-integrate-the-bloody-generated-code-with-my-own-code problem. Any one of these problems may be a showstopper; together, they make code generation an unlikely solution for most programming challenges.

Wouldn't it be great if we could realize the theoretical benefits of code generation—quicker, easier development, reduced redundancy, fewer bugs—without the drawbacks? That's what Andrei's approach promises. Generic components implement good designs in easy-to-use, mixable-and-matchable templates. They do pretty much what code generators do: produce boilerplate code for compiler consumption. The difference is that they do it within C++, not apart from it. The result is seamless integration with application code.

You can also use the full power of the language to extend, override, and otherwise tweak the designs to suit your needs.

Some of the techniques herein are admittedly tricky to grasp, especially the template metaprogramming in Chapter 3. Once you've mastered that, however, you'll have a solid foundation for the edifice of generic componentry, which almost builds itself in the ensuing chapters. In fact, I would argue that the metaprogramming material of Chapter 3 alone is worth the book's price—and there are ten other chapters full of insights to profit from. "Ten" represents an order of magnitude. Even so, the return on your investment will be far greater.

John Vlissides
IBM T.J. Watson Research
September 2000

Preface

You might be holding this book in a bookstore, asking yourself whether you should buy it. Or maybe you are in your employer's library, wondering whether you should invest time in reading it. I know you don't have time, so I'll cut to the chase. If you have ever asked yourself how to write higher-level programs in C++, how to cope with the avalanche of irrelevant details that plague even the cleanest design, or how to build reusable components that you don't have to hack into each time you take them to your next application, then this book is for you.

Imagine the following scenario. You come from a design meeting with a couple of printed diagrams, scribbled with your annotations. Okay, the event type passed between these objects is not `char` anymore; it's `int`. You change one line of code. The smart pointers to `Widget` are too slow; they should go unchecked. You change one line of code. The object factory needs to support the new `Gadget` class just added by another department. You change one line of code.

You have changed the design. Compile. Link. Done.

Well, there is something wrong with this scenario, isn't there? A much more likely scenario is this: You come from the meeting in a hurry because you have a pile of work to do. You fire a global search. You perform surgery on code. You add code. You introduce bugs. You remove the bugs . . . that's the way a programmer's job is, right? Although this book cannot possibly promise you the first scenario, it is nonetheless a resolute step in that direction. It tries to present C++ as a newly discovered language for software architects.

Traditionally, code is the most detailed and intricate aspect of a software system. Historically, in spite of various levels of language support for design methodologies (such as object orientation), a significant gap has persisted between the blueprints of a program and its code because the code must take care of the ultimate details of the implementation and of many ancillary tasks. The intent of the design is, more often than not, dissolved in a sea of quirks.

This book presents a collection of reusable design artifacts, called *generic components,* together with the techniques that make them possible. These generic components bring their users the well-known benefits of libraries, but in the broader space of system architecture. The coding techniques and the implementations provided focus on tasks and issues

that traditionally fall in the area of design, activities usually done *before* coding. Because of their high level, generic components make it possible to map intricate architectures to code in unusually expressive, terse, and easy-to-maintain ways.

Three elements are reunited here: design patterns, generic programming, and C++. These elements are combined to achieve a very high rate of reuse, both in the horizontal and vertical "markets" of software design. In the horizontal dimension, a small amount of library code implements a combinatorial—and essentially open-ended—number of structures and behaviors. On the vertical dimension, the generality of these components makes them applicable to a vast range of programs.

This book owes much to design patterns, powerful solutions to ever-recurring problems in object-oriented development. Design patterns are distilled pieces of good design—recipes for sound, reusable solutions to problems that can be encountered in many contexts. Design patterns concentrate on providing a suggestive lexicon for designs to be conveyed. They describe the problem, a time-proven solution with its variants, and the consequences of choosing each variant of that solution. Design patterns go above and beyond anything a programming language, no matter how advanced, could possibly express. By following and combining certain design patterns, the components presented in this book tend to address a large category of concrete problems.

Generic programming is a paradigm that focuses on abstracting types to a narrow collection of functional requirements and on implementing algorithms in terms of these requirements. Because algorithms define a strict and narrow interface to the types they operate on, the same algorithm can be used against a wide collection of types. The implementations in this book use generic programming techniques to achieve a minimal commitment to specificity, extraordinary terseness, and efficiency that rivals carefully handcrafted code.

C++ is the only implementation tool used in this book. You will not find in this book code that implements nifty windowing systems, complex networking libraries, or clever logging mechanisms. Instead, you will find the fundamental components that make it easy to implement all of the above, and much more. C++ has the breadth necessary to make this possible. Its underlying C memory model ensures raw performance, its support for polymorphism enables object-oriented techniques, and its templates unleash an incredible code generation machine. Templates pervade all the code in the book because they allow close cooperation between the user and the library. The user of the library literally controls the way code is generated, in ways constrained by the library. The role of a generic component library is to allow user-specified types and behaviors to be combined with generic components in a sound design. Because of the static nature of the techniques used, errors in mixing and matching the appropriate pieces are usually caught during compilation.

This book's manifest intent is to create generic components—preimplemented pieces of design whose main characteristics are flexibility, versatility, and ease of use. Generic components do not form a framework. In fact, their approach is complementary—whereas a framework defines interdependent classes to foster a specific object model, generic components are lightweight design artifacts that are independent of each other, yet can be mixed and matched freely. They can be of great help in *implementing* frameworks.

Audience

The intended audience of this book falls into two main categories. The first category is that of experienced C++ programmers who want to master the most modern library writing techniques. The book presents new, powerful C++ idioms that have surprising capabilities, some of which weren't even thought possible. These idioms are of great help in writing high-level libraries. Intermediate C++ programmers who want to go a step further will certainly find the book useful, too, especially if they invest a bit of perseverance. Although pretty hard-core C++ code is sometimes presented, it is thoroughly explained.

The second category consists of busy programmers who need to get the job done without undergoing a steep learning investment. They can skim the most intricate details of implementation and concentrate on *using* the provided library. Each chapter has an introductory explanation and ends with a Quick Facts section. Programmers will find these features a useful reference in understanding and using the components. The components can be understood in isolation, are very powerful yet safe, and are a joy to use.

You need to have a solid working experience with C++ and, above all, the desire to learn more. A degree of familiarity with templates and the Standard Template Library (STL) is desirable.

Having an acquaintance with design patterns (Gamma et al. 1995) is recommended but not mandatory. The patterns and idioms applied in the book are described in detail. However, this book is not a pattern book—it does not attempt to treat patterns in full generality. Because patterns are presented from the pragmatic standpoint of a library writer, even readers interested mostly in patterns may find the perspective refreshing, if constrained.

Loki

The book describes an actual C++ library called Loki. Loki is the god of wit and mischief in Norse mythology, and the author's hope is that the library's originality and flexibility will remind readers of the playful Norse god. All the elements of the library live in the namespace `Loki`. The namespace is not mentioned in the coding examples because it would have unnecessarily increased indentation and the size of the examples. Loki is freely available; you can download it from http://www.modernpcdesign.com.

Except for its threading part, Loki is written exclusively in standard C++. This, alas, means that many current compilers cannot cope with parts of it. I implemented and tested Loki using Metrowerks' CodeWarrior Pro 6.0 and Comeau C++ 4.2.38, both on Windows. It is likely that KAI C++ wouldn't have any problem with the code, either. As vendors release new, better compiler versions, you will be able to exploit everything Loki has to offer.

Loki's code and the code samples presented throughout the book use a popular coding standard originated by Herb Sutter. I'm sure you will pick it up easily. In a nutshell,

- Classes, functions, and enumerated types look `LikeThis`.
- Variables and enumerated values look `likeThis`.
- Member variables look `likeThis_`.
- Template parameters are declared with `class` if they can be only a user-defined type, and with `typename` if they can also be a primitive type.

Organization

The book consists of two major parts: techniques and components. Part I (Chapters 1 to 4) describes the C++ techniques that are used in generic programming and in particular in building generic components. A host of C++-specific features and techniques are presented: policy-based design, partial template specialization, typelists, local classes, and more. You may want to read this part sequentially and return to specific sections for reference.

Part II builds on the foundation established in Part I by implementing a number of generic components. These are not toy examples; they are industrial-strength components used in real-world applications. Recurring issues that C++ developers face in their day-to-day activity, such as smart pointers, object factories, and functor objects, are discussed in depth and implemented in a generic way. The text presents implementations that address basic needs and solve fundamental problems. Instead of explaining what a body of code does, the approach of the book is to discuss problems, take design decisions, and implement those decisions gradually.

Chapter 1 presents policies—a C++ idiom that helps in creating flexible designs.
Chapter 2 discusses general C++ techniques related to generic programming.
Chapter 3 implements typelists, which are powerful type manipulation structures.
Chapter 4 introduces an important ancillary tool: a small object allocator.
Chapter 5 introduces the concept of generalized functors, useful in designs that use the Command design pattern.
Chapter 6 describes Singleton objects.
Chapter 7 discusses and implements smart pointers.
Chapter 8 describes generic object factories.
Chapter 9 treats the Abstract Factory design pattern and provides implementations of it.
Chapter 10 implements several variations of the Visitor design pattern in a generic manner.
Chapter 11 implements several multimethod engines, solutions that foster various trade-offs.

The design themes cover many important situations that C++ programmers have to cope with on a regular basis. I personally consider object factories (Chapter 8) a cornerstone of virtually any quality polymorphic design. Also, smart pointers (Chapter 7) are an important component of many C++ applications, small and large. Generalized functors (Chapter 5) have an incredibly broad range of applications. Once you have generalized functors, many complicated design problems become very simple. The other, more specialized, generic components, such as Visitor (Chapter 10) or multimethods (Chapter 11), have important niche applications and stretch the boundaries of language support.

Acknowledgments

I would like to thank my parents for diligently taking care of the longest, toughest part of them all.

It should be stressed that this book, and much of my professional development, wouldn't have existed without Scott Meyers. Since we met at the C++ World Conference in 1998, Scott has constantly helped me do more and do better. Scott was the first person who enthusiastically encouraged me to develop my early ideas. He introduced me to John Vlissides, catalyzing another fruitful cooperation; lobbied Herb Sutter to accept me as a columnist for *C++ Report;* and introduced me to Addison-Wesley, practically forcing me into starting this book, at a time when I still had trouble understanding New York sales clerks. Ultimately, Scott helped me all the way through the book with reviews and advice, sharing with me all the pains of writing, and none of the benefits.

Many thanks to John Vlissides, who, with his sharp insights, convinced me of the problems with my solutions and suggested much better ones. Chapter 9 exists because John insisted that "things could be done better."

Thanks to P. J. Plauger and Marc Briand for encouraging me to write for the *C/C++ Users Journal,* at a time when I still believed that magazine columnists were inhabitants of another planet.

I am indebted to my editor, Debbie Lafferty, for her continuous support and sagacious advice.

My colleagues at RealNetworks, especially Boris Jerkunica and Jim Knaack, helped me very much by fostering an atmosphere of freedom, emulation, and excellence. I am grateful to them all for that.

I also owe much to all participants in the comp.lang.c++.moderated and comp.std.c++ Usenet newsgroups. These people greatly and generously contributed to my understanding of C++.

I would like to address thanks to the reviewers of early drafts of the manuscript: Mihail Antonescu, Bob Archer (my most thorough reviewer of all), Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginean, Patrick McKillen, Florin Mihaila, Sorin Oprea,

John Potter, Adrian Rapiteanu, Monica Rapiteanu, Brian Stanton, Adrian Steflea, Herb Sutter, John Torjo, Florin Trofin, and Cristi Vlasceanu. They all have invested significant efforts in reading and providing input, without which this book wouldn't have been half of what it is.

Thanks to Greg Comeau for providing me with his top-notch C++ compiler for free.

Last but not least, I would like to thank all my family and friends for their continuous encouragement and support.

7

Smart Pointers

Smart pointers have been the subject of hecatombs of code written and rivers of ink consumed by programmers and writers around the world. Perhaps the most popular, intricate, and powerful C++ idiom, smart pointers are interesting in that they combine many syntactic and semantic issues. This chapter discusses smart pointers, from their simplest aspects to their most complex ones and from the most obvious errors in implementing them to the subtlest ones—some of which also happen to be the most gruesome.

In brief, smart pointers are C++ objects that simulate simple pointers by implementing `operator->` and the unary `operator*`. In addition to sporting pointer syntax and semantics, smart pointers often perform useful tasks—such as memory management or locking—under the covers, thus freeing the application from carefully managing the lifetime of pointed-to objects.

This chapter not only discusses smart pointers but also implements a `SmartPtr` class template. `SmartPtr` is designed around policies (see Chapter 1), and the result is a smart pointer that has the exact levels of safety, efficiency, and ease of use that you want.

After reading this chapter, you will be an expert in smart pointer issues such as the following:

- The advantages and disadvantages of smart pointers
- Ownership management strategies
- Implicit conversions
- Tests and comparisons
- Multithreading issues

This chapter implements a generic `SmartPtr` class template. Each section presents one implementation issue in isolation. At the end, the implementation puts all the pieces together. In addition to understanding the design rationale of `SmartPtr`, you will know how to use, tweak, and extend it.

7.1 Smart Pointers 101

So what's a smart pointer? A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but it does more. Because smart pointers to different types

of objects tend to have a lot of code in common, almost all good-quality smart pointers in existence are templated by the pointee type, as you can see in the following code:

```
template <class T>
class SmartPtr
{
public:
   explicit SmartPtr(T* pointee) : pointee_(pointee) {...}
   SmartPtr& operator=(const SmartPtr& other);
   ~SmartPtr();
   T& operator*() const
   {
      ...
      return *pointee_;
   }
   T* operator->() const
   {
      ...
      return pointee_;
   }
private:
   T* pointee_;
   ...
};
```

`SmartPtr<T>` aggregates a pointer to `T` in its member variable `pointee_`. That's what most smart pointers do. In some cases, a smart pointer might aggregate some handles to data and compute the pointer on the fly.

The two operators give `SmartPtr` pointer-like syntax and semantics. That is, you can write

```
class Widget
{
public:
   void Fun();
};

SmartPtr<Widget> sp(new Widget);
sp->Fun();
(*sp).Fun();
```

Aside from the definition of `sp`, nothing reveals it as not being a pointer. This is the mantra of smart pointers: You can replace pointer definitions with smart pointer definitions without incurring major changes to your application's code. You thus get extra goodies with ease. Minimizing code changes is very appealing and vital for getting large applications to use smart pointers. As you will soon see, however, smart pointers are not a free lunch.

7.2 The Deal

But what's the deal with smart pointers? you might ask. What do you gain by replacing simple pointers with smart pointers? The explanation is simple. Smart pointers have value semantics, whereas some simple pointers do not.

An object with value semantics is an object that you can *copy* and *assign to.* A plain `int` is the perfect example of a first-class object. You can create, copy, and change integer values freely. A pointer that you use to iterate in a buffer also has value semantics—you initialize it to point to the beginning of the buffer, and you bump it until you reach the end. Along the way, you can copy its value to other variables to hold temporary results.

With pointers that hold values allocated with `new`, however, the story is very different. Once you have written

```
Widget* p = new Widget;
```

the variable `p` not only points to, but also *owns,* the memory allocated for the `Widget` object. This is because later you must issue `delete p` to ensure that the `Widget` object is destroyed and its memory is released. If in the line after the line just shown you write

```
p = 0; // assign something else to p
```

you lose ownership of the object previously pointed to by `p`, and you have no chance at all to get a grip on it again. You have a resource leak, and resource leaks never help.

Furthermore, when you copy `p` into another variable, the compiler does not automatically manage the ownership of the memory to which the pointer points. All you get is two raw pointers pointing to the same object, and you have to track them even more carefully because double deletions are even more catastrophic than no deletion. Consequently, pointers to allocated objects do *not* have value semantics—you cannot copy and assign to them at will.

Smart pointers can be of great help in this area. Most smart pointers offer *ownership management* in addition to pointer-like behavior. Smart pointers can figure out how ownership evolves, and their destructors can release the memory according to a well-defined strategy. Many smart pointers hold enough information to take full initiative in releasing the pointed-to object.

Smart pointers may manage ownership in various ways, each appropriate to a category of problems. Some smart pointers transfer ownership automatically: After you copy a smart pointer to an object, the source smart pointer becomes null, and the destination points to (and holds ownership of) the object. This is the behavior implemented by the standard-provided `std::auto_ptr`. Other smart pointers implement reference counting: They track the total count of smart pointers that point to the same object, and when this count goes down to zero, they `delete` the pointed-to object. Finally, some others duplicate their pointed-to object whenever you copy them.

In short, in the smart pointers' world, ownership is an important topic. By providing ownership management, smart pointers are able to support integrity guarantees and full value semantics. Because ownership has much to do with constructing, copying, and destroying smart pointers, it's easy to figure out that these are the most vital functions of a smart pointer.

The following few sections discuss various aspects of smart pointer design and implementation. The goal is to render smart pointers as close to raw pointers as possible, but not closer. It's a contradictory goal: After all, if your smart pointers behave *exactly* like dumb pointers, they *are* dumb pointers.

In implementing compatibility between smart pointers and raw pointers, there is a thin line between nicely filling compatibility checklists and paving the way to chaos. You will find that adding seemingly worthwhile features might expose the clients to costly risks. Much of the craft of implementing good smart pointers consists of carefully balancing their set of features.

7.3 Storage of Smart Pointers

To start, let's ask a fundamental question about smart pointers. Is `pointee_`'s type necessarily `T*`? If not, what else could it be? In generic programming, you should always ask yourself questions like these. Each type that's hardcoded in a piece of generic code decreases the genericity of the code. Hardcoded types are to generic code what magic constants are to regular code.

In several situations, it is worthwhile to allow customizing the pointee type. One situation is when you deal with nonstandard pointer modifiers. In the 16-bit Intel 80x86 days, you could qualify pointers with modifiers like `__near`, `__far`, and `__huge`. Other segmented memory architectures use similar modifiers.

Another situation is when you want to layer smart pointers. What if you have a `LegacySmartPtr<T>` smart pointer implemented by someone else, and you want to enhance it? Would you derive from it? That's a risky decision. It's better to wrap the legacy smart pointer into a smart pointer of your own. This is possible because the inner smart pointer supports pointer syntax. From the outer smart pointer's viewpoint, the pointee type is not `T*` but `LegacySmartPtr<T>`.

There are interesting applications of smart pointer layering, mainly because of the mechanics of `operator->`. When you apply `operator->` to a type that's not a built-in pointer, the compiler does an interesting thing. After looking up and applying the user-defined `operator->` to that type, it applies `operator->` again to the result. The compiler keeps doing this recursively until it reaches a native pointer, and only then proceeds with member access. It follows that a smart pointer's `operator->` does not have to return a pointer. It can return an object that in turn implements `operator->`, without changing the use syntax.

This leads to a very interesting idiom: pre- and postfunction calls (Stroustrup 2000). If you return an object of type `PointerType` by value from `operator->`, the sequence of execution is as follows:

1. Constructor of `PointerType`
2. `PointerType::operator->` called; likely returns a pointer to an object of type `PointeeType`
3. Member access for `PointeeType`—likely a function call
4. Destructor of `PointerType`

In a nutshell, you have a nifty way of implementing locked function calls. This idiom has broad uses with multithreading and locked resource access. You can have `PointerType`'s constructor lock the resource, and then you can access the resource; finally, `PointerType`'s destructor unlocks the resource.

The generalization doesn't stop here. The syntax-oriented "pointer" part might some-

times pale in comparison with the powerful resource management techniques that are included in smart pointers. It follows that, in rare cases, smart pointers could drop the pointer syntax. An object that does not define `operator->` and `operator*` violates the definition of a smart pointer, but there are objects that do deserve smart pointer–like treatment, although they are not, strictly speaking, smart pointers.

Look at real-world APIs and applications. Many operating systems foster *handles* as accessors to certain internal resources, such as windows, mutexes, or devices. Handles are intentionally obfuscated pointers; one of their purposes is to prevent their users from manipulating critical operating system resources directly. Most of the time, handles are integral values that are indices in a hidden table of pointers. The table provides the additional level of indirection that protects the inner system from the application programmers. Although they don't provide an `operator->`, handles resemble pointers in semantics and in the way they are managed.

For such a smart resource, it does not make sense to provide `operator->` or `operator*`. However, you do take advantage of all the resource management techniques that are specific to smart pointers.

To generalize the type universe of smart pointers, we distinguish three potentially distinct types in a smart pointer:

- *The storage type.* This is the type of `pointee_`. By "default"—in regular smart pointers—it is a raw pointer.
- *The pointer type.* This is the type returned by `operator->`. It can be different from the storage type if you want to return a proxy object instead of just a pointer. (You will find an example of using proxy objects later in this chapter.)
- *The reference type.* This is the type returned by `operator*`.

It would be useful if `SmartPtr` supported this generalization in a flexible way. Thus, the three types mentioned here ought to be abstracted in a policy called Storage.

In conclusion, smart pointers can, and should, generalize their pointee type. To do this, `SmartPtr` abstracts three types in a Storage policy: the stored type, the pointer type, and the reference type. Not all types necessarily make sense for a given `SmartPtr` instantiation. Therefore, in rare cases (handles), a policy might disable access to `operator->` or `operator*` or both.

7.4 Smart Pointer Member Functions

Many existing smart pointer implementations allow operations through member functions, such as `Get` for accessing the pointee object, `Set` for changing it, and `Release` for taking over ownership. This is the obvious and natural way of encapsulating `SmartPtr`'s functionality.

However, experience has proven that member functions are not very suitable for smart pointers. The reason is that the interaction between member function calls for the smart pointer and for the *pointed-to* object can be extremely confusing.

Suppose, for instance, that you have a `Printer` class with member functions such as `Acquire` and `Release`. With `Acquire` you take ownership of the printer so that no other application prints to it, and with `Release` you relinquish ownership. As you use a smart

pointer to Printer, you may notice a strange syntactical closeness to things that are very far apart semantically.

```
SmartPtr<Printer> spRes = ...;
spRes->Acquire(); // acquire the printer
... print a document ...
spRes->Release(); // release the printer
spRes.Release();  // release the pointer to the printer
```

The user of SmartPtr now has access to two totally different worlds: the world of the pointed-to object members and the world of the smart pointer members. A matter of a dot or an arrow thinly separates the two worlds.

On the face of it, C++ does force you routinely to observe certain slight differences in syntax. A Pascal programmer learning C++ might even feel that the slight syntactic difference between & and && is an abomination. Yet C++ programmers don't even blink at it. They are trained by habit to distinguish such syntax matters easily.

However, smart pointer member functions defeat training by habit. Raw pointers don't have member functions, so C++ programmers' eyes are not habituated to detect and distinguish dot calls from arrow calls. The compiler does a good job at that: If you use a dot after a raw pointer, the compiler will yield an error. Therefore, it is easy to imagine, and experience proves, that even seasoned C++ programmers find it extremely disturbing that both sp.Release() and sp->Release() compile flag-free but do very different things. The cure is simple: A smart pointer should not use member functions. SmartPtr uses only nonmember functions. These functions become friends of the smart pointer class.

Overloaded functions can be just as confusing as member functions of smart pointers, but there is an important difference. C++ programmers already use overloaded functions. Overloading is an important part of the C++ language and is used routinely in library and application development. This means that C++ programmers *do* pay attention to differences in function call syntax—such as Release(*sp) versus Release(sp)—in writing and reviewing code.

The only functions that necessarily remain members of SmartPtr are the constructors, the destructor, operator=, operator->, and unary operator*. All other operations of SmartPtr are provided through named nonmember functions.

For reasons of clarity, SmartPtr does not have any named member functions. The only functions that access the pointee object are GetImpl, GetImplRef, Reset, and Release, which are defined at the namespace level.

```
template <class T> T* GetImpl(SmartPtr<T>& sp);
template <class T> T*& GetImplRef(SmartPtr<T>& sp);
template <class T> void Reset(SmartPtr<T>& sp, T* source);
template <class T> void Release(SmartPtr<T>& sp, T*& destination);
```

- GetImpl returns the pointer object stored by SmartPtr.
- GetImplRef returns *a reference* to the pointer object stored by SmartPtr. GetImplRef allows you to change the underlying pointer, so it requires extreme care in use.
- Reset resets the underlying pointer to another value, releasing the previous one.
- Release releases ownership of the smart pointer, giving its user the responsibility of managing the pointee object's lifetime.

The actual declarations of these four functions in Loki are slightly more elaborate. They don't assume that the type of the pointer object stored by `SmartPtr` is `T*`. As discussed in Section 7.3, the Storage policy defines the storage type. Most of the time, it's a straight pointer, except in exotic implementations of Storage, when it might be a handle or an elaborate type.

7.5 Ownership-Handling Strategies

Ownership handling is often the most important *raison d'être* of smart pointers. Usually, from their clients' viewpoint, smart pointers own the objects to which they point. A smart pointer is a first-class value that takes care of deleting the pointed-to object under the covers. The client can intervene in the pointee object's lifetime by issuing calls to helper management functions.

For implementing self-ownership, smart pointers must carefully track the pointee object, especially during copying, assignment, and destruction. Such tracking brings some overhead in space, time, or both. An application should settle on the strategy that best fits the problem at hand and does not cost too much.

The following subsections discuss the most popular ownership management strategies and how `SmartPtr` implements them.

7.5.1 Deep Copy

The simplest strategy applicable is to copy the pointee object whenever you copy the smart pointer. If you ensure this, there is only one smart pointer for each pointee object. Therefore, the smart pointer's destructor can safely `delete` the pointee object. Figure 7.1 depicts the state of affairs if you use smart pointers with deep copy.

At first glance, the deep copy strategy sounds rather dull. It seems as if the smart pointer does not add any value over regular C++ value semantics. Why would you make the effort of using a smart pointer, when simple pass by value of the pointee object works just as well?

The answer is *support for polymorphism.* Smart pointers are vehicles for transporting polymorphic objects safely. You hold a smart pointer to a base class, which might actually point to a derived class. When you copy the smart pointer, you want to copy its polymorphic behavior, too. It's interesting that you don't exactly know what behavior and state you are dealing with, but you certainly need to duplicate that behavior and state.

Because deep copy most often deals with polymorphic objects, the following naive implementation of the copy constructor is wrong:

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(const SmartPtr& other)
   : pointee_(new T(*other.pointee_))
   {
   }
   ...
};
```

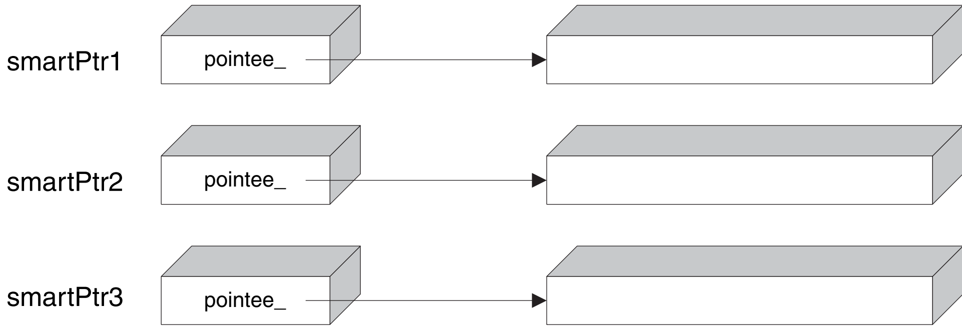

Figure 7.1: Memory layout for smart pointers with deep copy

Say you copy an object of type `SmartPtr<Widget>`. If `other` points to an instance of a class `ExtendedWidget` that derives from `Widget`, the copy constructor above copies only the `Widget` part of the `ExtendedWidget` object. This phenomenon is known as *slicing*—only the `Widget` "slice" of the object of the presumably larger type `ExtendedWidget` gets copied. Slicing is most often undesirable. It is a pity that C++ allows slicing so easily—a simple call by value slices objects without any warning.

Chapter 8 discusses cloning in depth. As shown there, the classic way of obtaining a polymorphic clone for a hierarchy is to define a virtual `Clone` function and implement it as follows:

```
class AbstractBase
{
   ...
   virtual AbstractBase* Clone() = 0;
};

class Concrete : public AbstractBase
{
   ...
   virtual AbstractBase* Clone()
   {
      return new Concrete(*this);
   }
};
```

The `Clone` implementation must follow the same pattern in all derived classes; in spite of its repetitive structure, there is no reasonable way to automate defining the `Clone` member function (beyond macros, that is).

A generic smart pointer cannot count on knowing the exact name of the cloning member function—maybe it's `clone`, or maybe `MakeCopy`. Therefore, the most flexible approach is to parameterize `SmartPtr` with a policy that addresses cloning.

7.5.2 Copy on Write

Copy on write (COW, as it is fondly called by its fans) is an optimization technique that avoids unnecessary object copying. The idea that underlies COW is to clone the pointee object at the first attempt of modification; until then, several pointers can share the same object.

Smart pointers, however, are not the best place to implement COW, because smart pointers cannot differentiate between calls to `const` and non-`const` member functions of the pointee object. Here is an example:

```
template <class T>
class SmartPtr
{
public:
   T* operator->() { return pointee_; }
   ...
};

class Foo
{
public:
   void ConstFun() const;
   void NonConstFun();
};

...
SmartPtr<Foo> sp;
sp->ConstFun(); // invokes operator->, then ConstFun
sp->NonConstFun(); // invokes operator->, then NonConstFun
```

The same `operator->` gets invoked for both functions called; therefore, the smart pointer does not have any clue whether to make the COW or not. Function invocations for the pointee object happen somewhere beyond the reach of the smart pointer. (Section 7.11 explains how `const` interacts with smart pointers and the objects they point to.)

In conclusion, COW is effective mostly as an implementation optimization for full-featured classes. Smart pointers are at too low a level to implement COW semantics effectively. Of course, smart pointers can be good building blocks in implementing COW for a class.

The `SmartPtr` implementation in this chapter does not provide support for COW.

7.5.3 Reference Counting

Reference counting is the most popular ownership strategy used with smart pointers. Reference counting tracks the number of smart pointers that point to the same object. When that number goes to zero, the pointee object is deleted. This strategy works very well if you don't break certain rules—for instance, you should not keep dumb pointers and smart pointers to the same object.

The actual counter must be shared among smart pointer objects, leading to the structure

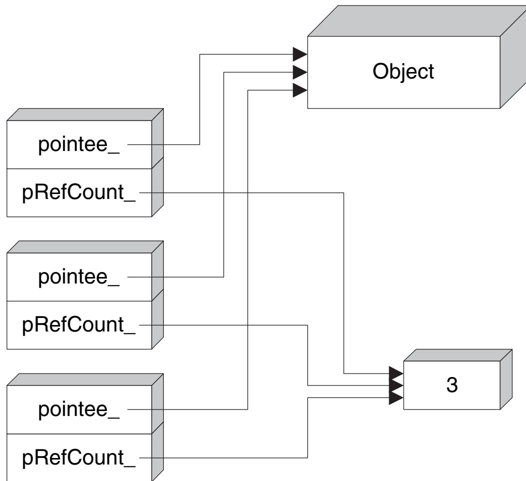

Figure 7.2: Three reference-counted smart pointers pointing to the same object

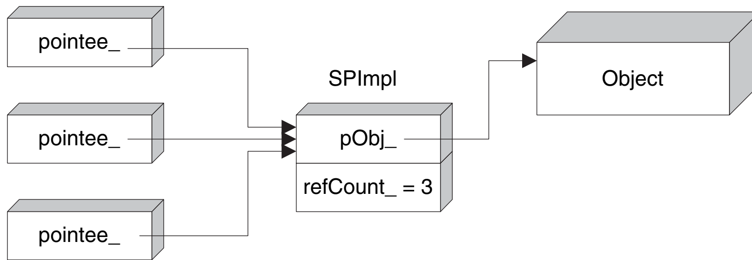

Figure 7.3: An alternate structure of reference-counted pointers

depicted in Figure 7.2. Each smart pointer holds a pointer to the reference counter (`pRefCount_` in Figure 7.2) in addition to the pointer to the object itself. This usually doubles the size of the smart pointer, which may or may not be an acceptable amount of overhead, depending on your needs and constraints.

There is another, subtler overhead issue. Reference-counted smart pointers must store the reference counter on the free store. The problem is that in many implementations, the default C++ free store allocator is remarkably slow and wasteful of space when it comes to allocating small objects, as discussed in Chapter 4. (Obviously, the reference count, typically occupying 4 bytes, does qualify as a small object.) The overhead in speed stems from slow algorithms in finding available chunks of memory, and the overhead in size is incurred by the bookkeeping information that the allocator holds for each chunk.

The relative size overhead can be partially mitigated by holding the pointer and the reference count together, as in Figure 7.3. The structure in Figure 7.3 reduces the size of the smart pointer to that of a pointer, but at the expense of access speed: The pointee object is

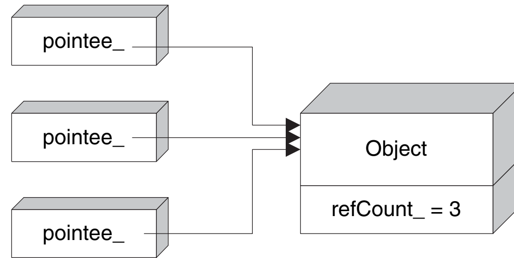

Figure 7.4: Intrusive reference counting

an extra level of indirection away. This is a considerable drawback because you typically use a smart pointer several times, whereas you obviously construct and destroy it only once.

The most efficient solution is to hold the reference counter in the pointee object itself, as shown in Figure 7.4. This way `SmartPtr` is just the size of a pointer, and there is no extra overhead at all. This technique is known as *intrusive reference counting,* because the reference count is an "intruder" in the pointee—it semantically belongs to the smart pointer. The name also gives a hint about the Achilles' heel of the technique: You must design up front or modify the pointee class to support reference counting.

A generic smart pointer should use intrusive reference counting where available and implement a nonintrusive reference counting scheme as an acceptable alternative. For implementing nonintrusive reference counting, the small-object allocator presented in Chapter 4 can help a great deal. The `SmartPtr` implementation using nonintrusive reference counting leverages the small-object allocator, thus slashing the performance overhead caused by the reference counter.

7.5.4 Reference Linking

Reference linking relies on the observation that you don't really need the actual count of smart pointer objects pointing to one pointee object; you only need to detect when that count goes down to zero. This leads to the idea of keeping an "ownership list," as shown in Figure 7.5.[1]

All `SmartPtr` objects that point to a given pointee form a doubly linked list. When you create a new `SmartPtr` from an existing `SmartPtr`, the new object is appended to the list; `SmartPtr`'s destructor takes care of removing the destroyed object from the list. When the list becomes empty, the pointee object is deleted.

The doubly linked list structure fits reference linking like a glove. You cannot use a singly linked list because removals from such a list take linear time. You cannot use a vector because the `SmartPtr` objects are not contiguous (and removals from vectors take linear time anyway). You need a structure sporting constant-time append, constant-time remove, and constant-time empty detection. This bill is fit precisely and exclusively by doubly linked lists.

[1] Risto Lankinen described the reference-linking mechanism on the Usenet in November 1995.

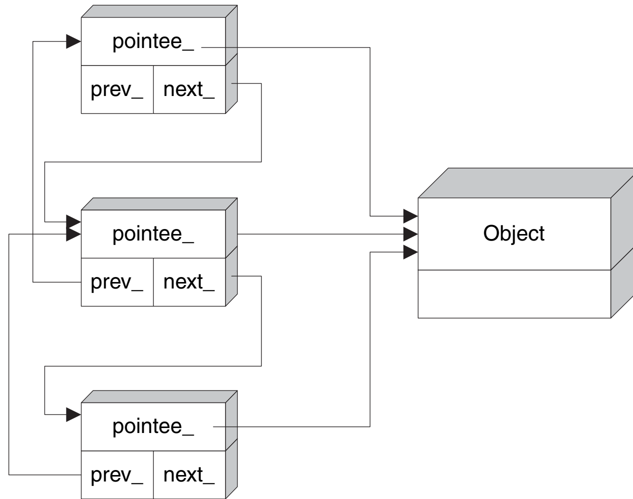

Figure 7.5: Reference linking in action

In a reference-linking implementation, each `SmartPtr` object holds two extra pointers—one to the next element and one to the previous element.

The advantage of reference linking over reference counting is that the former does not use extra free store, which makes it more reliable: Creating a reference-linked smart pointer cannot fail. The disadvantage is that reference linking needs more memory for its bookkeeping (three pointers versus only one pointer plus one integer). Also, reference counting should be a bit speedier—when you copy smart pointers, only an indirection and an increment are needed. The list management is slightly more elaborate. In conclusion, you should use reference linking only when the free store is scarce. Otherwise, prefer reference counting.

To wrap up the discussion on reference count management strategies, let's note a significant disadvantage that they have. Reference management—be it counting or linking—is a victim of the resource leak known as *cyclic reference.* Imagine an object `A` holds a smart pointer to an object `B`. Also, object `B` holds a smart pointer to `A`. These two objects form a cyclic reference; even though you don't use any of them anymore, they use each other. The reference management strategy cannot detect such cyclic references, and the two objects remain allocated forever. The cycles can span multiple objects, closing circles that often reveal unexpected dependencies that are very hard to debug.

In spite of this, reference management is a robust, speedy ownership-handling strategy. If used with precaution, reference management makes application development significantly easier.

7.5.5 Destructive Copy

Destructive copy does exactly what you think it does: During copying, it destroys the object being copied. In the case of smart pointers, destructive copy destroys the source smart

pointer by taking its pointee object and passing it to the destination smart pointer. The `std::auto_ptr` class template features destructive copy.

In addition to being suggestive about the action taken, "destructive" also vividly describes the dangers associated with this strategy. Misusing destructive copy may have destructive effects on your program data, your program correctness, and your brain cells.

Smart pointers may use destructive copy to ensure that at any time there is only one smart pointer pointing to a given object. During the copying or assignment of one smart pointer to another, the "living" pointer is passed to the destination of the copy, and the source's `pointee_` becomes zero. The following code illustrates a copy constructor and an assignment operator of a simple `SmartPtr` featuring destructive copy.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(SmartPtr& src)
   {
      pointee_ = src.pointee_;
      src.pointee_ = 0;
   }
   SmartPtr& operator=(SmartPtr& src)
   {
      if (this != &src)
      {
         delete pointee_;
         pointee_ = src.pointee_;
         src.pointee_ = 0;
      }
      return *this;
   }
   ...
};
```

C++ etiquette calls for the right-hand side of the copy constructor and the assignment operator to be a reference to a `const` object. Classes that foster destructive copy break this convention for obvious reasons. Because etiquette exists for a reason, you should expect negative consequences if you break it. Indeed, here they are:

```
void Display(SmartPtr<Something> sp);
...
SmartPtr<Something> sp(new Something);
Display(sp);  // sinks sp
```

Although `Display` means no harm to its argument (accepts it by value), it acts like a maelstrom of smart pointers: It sinks any smart pointer passed to it. After `Display(sp)` is called, `sp` holds the null pointer.

Because they do not support value semantics, smart pointers with destructive copy cannot be stored in standard containers and in general must be handled with almost as much care as raw pointers.

The ability to store smart pointers in a container is very important. Containers of raw

pointers make manual ownership management tricky, so many containers of pointers can use smart pointers to good advantage. Smart pointers with destructive copy, however, do not mix with containers.

On the bright side, smart pointers with destructive copy have significant advantages:

- They incur almost no overhead.
- They are good at enforcing ownership transfer semantics. In this case, you use the "maelstrom effect" described earlier to your advantage: You make it clear that your function takes over the passed-in pointer.
- They are good as return values from functions. If the smart pointer implementation uses a certain trick,[2] you can return smart pointers with destructive copy from functions. This way, you can be sure that the pointee object gets destroyed if the caller doesn't use the return value.
- They are excellent as stack variables in functions that have multiple return paths. You don't have to remember to delete the pointee object manually—the smart pointer takes care of this for you.

The destructive copy strategy is used by the standard-provided `std::auto_ptr`. This brings destructive copy another important advantage:

- Smart pointers with destructive copy semantics are the only smart pointers that the standard provides, which means that many programmers will get used to their behavior sooner or later.

For these reasons, the `SmartPtr` implementation should provide optional support for destructive copy semantics.

Smart pointers use various ownership semantics, each having its own trade-offs. The most important techniques are deep copy, reference counting, reference linking, and destructive copy. `SmartPtr` implements all these strategies through an Ownership policy, allowing its users to choose the one that best fits an application's needs. The default strategy is reference counting.

7.6 The Address-of Operator

In striving to make smart pointers as indistinguishable as possible from their native counterparts, designers stumbled upon an obscure operator that is on the list of overloadable operators: unary `operator&`, the *address-of operator*.[3]

An implementer of smart pointers might choose to overload the address-of operator like this:

```
template <class T>
class SmartPtr
{
public:
```

[2] Invented by Greg Colvin and Bill Gibbons for `std::auto_ptr`.

[3] *Unary* `operator&` is to differentiate it from binary `operator&`, which is the bitwise AND operator.

```
   T** operator&()
   {
      return &pointee_;
   }
   ...
};
```

After all, if a smart pointer is to simulate a pointer, then its address must be substitutable for the address of a regular pointer. This overload makes code like the following possible:

```
void Fun(Widget** pWidget);
...
SmartPtr<Widget> spWidget(...);
Fun(&spWidget); // okay, invokes operator& and obtains a
                // pointer to pointer to Widget
```

It seems very desirable to have such an accurate compatibility between smart pointers and dumb pointers, but overloading the unary `operator&` is one of those clever tricks that can do more harm than good. There are two reasons why overloading unary `operator&` is not a very good idea.

One reason is that exposing the address of the pointed-to object implies giving up any automatic ownership management. When a client freely accesses the address of the raw pointer, any helper structures that the smart pointer holds, such as reference counts, become invalid for all purposes. While the client deals directly with the address of the raw pointer, the smart pointer is completely unconscious.

The second reason, a more pragmatic one, is that overloading unary `operator&` makes the smart pointer unusable with STL containers. Actually, overloading unary `operator&` for a type pretty much makes generic programming impossible for that type, because the address of an object is too fundamental a property to play with naively. Most generic code assumes that applying `&` to an object of type `T` returns an object of type `T*`—you see, address-of is a fundamental concept. If you defy this concept, generic code behaves strangely either at compile time or—worse—at runtime.

Thus, it is not recommended that unary `operator&` be overloaded for smart pointers or for any objects in general. `SmartPtr` does not overload unary `operator&`.

7.7 Implicit Conversion to Raw Pointer Types

Consider this code:

```
void Fun(Something* p);
...
SmartPtr<Something> sp(new Something);
Fun(sp); // OK or error?
```

Should this code compile or not? Following the “maximum compatibility” line of thought, the answer is yes.

Technically, it is very simple to render the previous code compilable by introducing a user-defined conversion:

```
template <class T>
class SmartPtr
{
public:
   operator T*() // user-defined conversion to T*
   {
      return pointee_;
   }
   ...
};
```

However, this is not the end of the story.

User-defined conversions in C++ have an interesting history. Back in the 1980s, when user-defined conversions were introduced, most programmers considered them a great invention. User-defined conversions promised a more unified type system, expressive semantics, and the ability to define new types that were indistinguishable from built-in ones. With time, however, user-defined conversions revealed themselves as awkward and potentially dangerous. They might become dangerous especially when they expose handles to internal data (Meyers 1998a, Item 29), which is precisely the case with the `operator T*` in the previous code. That's why you should think carefully before allowing automatic conversions for the smart pointers you design.

One potential danger comes inherently from giving the user unattended access to the raw pointer that the smart pointer wraps. Passing the raw pointer around defeats the inner workings of the smart pointer. Once unleashed from the confines of its wrapper, the raw pointer can easily become a threat to program sanity again, just as it was before smart pointers were introduced.

Another danger is that user-defined conversions pop up unexpectedly, even when you don't need them. Consider the following code:

```
SmartPtr<Something> sp;
...
// A gross semantic error
// However, it goes undetected at compile time
delete sp;
```

The compiler matches operator `delete` with the user-defined conversion to `T*`. At runtime, `operator T*` is called, and `delete` is applied to its result. This is certainly not what you want to do to a smart pointer, because it is supposed to manage ownership itself. An extra unwitting `delete` call throws out the window all the careful ownership management that the smart pointer performs under the covers.

There are quite a few ways to prevent the `delete` call from compiling. Some of them are very ingenious (Meyers 1996). One that's very effective and easy to implement is to make the call to `delete` intentionally *ambiguous.* You can achieve this by providing *two* automatic conversions to types that are susceptible to a call to `delete`. One type is `T*` itself, and the other can be `void*`.

```
template <class T>
class SmartPtr
{
public:
   operator T*() // User-defined conversion to T*
   {
      return pointee_;
   }
   operator void*() // Added conversion to void*
   {
      return pointee_;
   }
   ...
};
```

A call to `delete` against such a smart pointer object is ambiguous. The compiler cannot decide which conversion to apply, and the trick above exploits this indecision to good advantage.

Don't forget that disabling the `delete` operator was only a part of the issue. Whether to provide an automatic conversion to a raw pointer remains an important decision in implementing a smart pointer. It's too dangerous just to let it in, yet too convenient to rule it out. The final `SmartPtr` implementation will give you a choice about that.

However, forbidding implicit conversion does not necessarily eliminate all access to the raw pointer; it is often necessary to gain such access. Therefore, all smart pointers do provide *explicit* access to their wrapped pointer via a call to a function:

```
void Fun(Something* p);
...
SmartPtr<Something> sp;
Fun(GetImpl(sp)); // OK, explicit conversion always allowed
```

It's not a matter of whether you can get to the wrapped pointer; it's how easy it is. This may seem like a minor difference, but it's actually very important. An implicit conversion happens without the programmer or the maintainer noticing or even knowing it. An explicit conversion—as is the call to `GetImpl`—passes through the mind, the understanding, and the fingers of the programmer and remains written in the code for everybody to see it.

Implicit conversion from the smart pointer type to the raw pointer type is desirable, but sometimes dangerous. `SmartPtr` provides this implicit conversion as a choice. The default is on the safe side—no implicit conversions. Explicit access is always available through the `GetImpl` function.

7.8 Equality and Inequality

C++ teaches its users that any clever trick such as the one presented in the previous section (intentional ambiguity) establishes a new context, which in turn may have unexpected ripples.

Consider tests for equality and inequality of smart pointers. A smart pointer should

support the same comparison syntax that raw pointers support. Programmers expect the following tests to compile and run as they do for a raw pointer.

```
SmartPtr<Something> sp1, sp2;
Something* p;
...
if (sp1)    // Test 1: direct test for non-null pointer
   ...
if (!sp1)      // Test 2: direct test for null pointer
   ...
if (sp1 == 0)   // Test 3: explicit test for null pointer
   ...
if (sp1 == sp2) // Test 4: comparison of two smart pointers
   ...
if (sp1 == p)   // Test 5: comparison with a raw pointer
   ...
```

There are more tests than depicted here if you consider symmetry and `operator!=`. If we solve the equality tests, we can easily define the corresponding symmetric and inequality tests.

There is an unfortunate interference between the solution to the previous issue (preventing `delete` from compiling) and a possible solution to this issue. With one user-defined conversion to the raw pointer type, most of the test expressions (except test 4) compile successfully and run as expected. The downside is that you can accidentally call the `delete` operator against the smart pointer. With two user-defined conversions (intentional ambiguity), you detect wrongful `delete` calls, but none of these tests compiles anymore—they have become ambiguous too.

An additional user-defined conversion to `bool` helps, but this, to nobody's surprise, introduces new trouble. Given this smart pointer:

```
template <class T>
class SmartPtr
{
public:
   operator bool() const
   {
      return pointee_ != 0;
   }
   ...
};
```

the four tests compile, but so do the following nonsensical operations:

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2; // Orange is unrelated to Apple
if (sp1 == sp2)       // Converts both pointers to bool
                      // and compares results
   ...
if (sp1 != sp2)       // Ditto
   ...
```

```
bool b = sp1;          // The conversion allows this, too
if (sp1 * 5 == 200)    // Ouch! SmartPtr behaves like an integral
                       // type!
   ...
```

As you can see, it's either not at all or too much: Once you add a user-defined conversion to `bool`, you allow `SmartPtr` to act as a `bool` in many more situations than you actually wanted. For all practical purposes, defining an `operator bool` for a smart pointer is not a smart solution.

A true, complete, rock-solid solution to this dilemma is to go all the way and overload each and every operator separately. This way any operation that makes sense for the bare pointer makes sense for the smart pointer, and nothing else. Here is the code that implements this idea.

```
template <class T>
class SmartPtr
{
public:
   bool operator!() const // Enables "if (!sp) ..."
   {
      return pointee_ == 0;
   }
   inline friend bool operator==(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   inline friend bool operator==(const T* lhs,
      const SmartPtr& rhs)
   {
      return lhs == rhs.pointee_;
   }
   inline friend bool operator!=(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ != rhs;
   }
   inline friend bool operator!=(const T* lhs,
      const SmartPtr& rhs)
   {
      return lhs != rhs.pointee_;
   }
   ...
};
```

Yes, it's a pain, but this approach solves the problems with almost all comparisons, including the tests against the literal zero. What the forwarding operators in this code do is to pass operators that client code applies to the smart pointer on to the raw pointer that the smart pointer wraps. No simulation can be more realistic than that.

We still haven't solved the problem completely. If you provide an automatic conversion

to the pointee type, there still is the risk of ambiguities. Suppose you have a class `Base` and a class `Derived` that inherits `Base`. Then the following code makes practical sense yet is ill formed due to ambiguity.

```
SmartPtr<Base> sp;
Derived* p;
...
if (sp == p) {} // error! Ambiguity between:
                // '(Base*)sp == (Base*)p'
                // and 'operator==(sp, (Base*)p)'
```

Indeed, smart pointer development is not for the faint of heart.

We're not out of bullets, though. In addition to the definitions of `operator==` and `operator!=`, we can add *templated* versions of them, as you can see in the following code:

```
template <class T>
class SmartPtr
{
public:
   ... as above ...
   template <class U>
   inline friend bool operator==(const SmartPtr& lhs,
      const U* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   template <class U>
   inline friend bool operator==(const U* lhs,
      const SmartPtr& rhs)
   {
      return lhs == rhs.pointee_;
   }
   ... similarly defined operator!= ...
};
```

The templated operators are "greedy" in the sense that they match comparisons with any pointer type whatsoever, thus consuming the ambiguity.

If that's the case, why should we keep the nontemplated operators—the ones that take the pointee type? They never get a chance to match, because the template matches any pointer type, including the pointee type itself.

The rule that "never" actually means "almost never" applies here, too. In the test `if (sp == 0)`, the compiler tries the following matches.

- *The templated operators.* They don't match because the type of literal zero is not a pointer type. A literal zero can be implicitly converted to a pointer type, but template matching does not include conversions.
- *The nontemplated operators.* After eliminating the templated operators, the compiler tries the nontemplated ones. One of these operators kicks in through an implicit conversion from the literal zero to the pointee type. Had the nontemplated operators not existed, the test would have been an error.

In conclusion, we need *both* the nontemplated and the templated comparison operators.

Let's see now what happens if we compare two `SmartPtr`s instantiated with different types.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)
   ...
```

The compiler chokes on the comparison because of an ambiguity: Each of the two `SmartPtr` instantiations defines an `operator==`, and the compiler does not know which one to choose. We can dodge this problem by defining an "ambiguity buster" as shown:

```
template <class T>
class SmartPtr
{
public:
   // Ambiguity buster
   template <class U>
   bool operator==(const SmartPtr<U>& rhs) const
   {
      return pointee_ == rhs.pointee_;
   }
   // Similarly for operator!=
   ...
};
```

This newly added operator is a member that specializes exclusively in comparing `SmartPtr<...>` objects. The beauty of this ambiguity buster is that it makes smart pointer comparisons act like raw pointer comparisons. If you compare two smart pointers to `Apple` and `Orange`, the code will be essentially equivalent to comparing two raw pointers to `Apple` and `Orange`. If the comparison makes sense, then the code compiles; otherwise, it's a compile-time error.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)   // Semantically equivalent to
                  // sp1.pointee_ == sp2.pointee_
   ...
```

There is one unsatisfied syntactic artifact left, namely, the direct test `if (sp)`. Here life becomes really interesting. The `if` statement applies only to expressions of arithmetic and pointer type. Consequently, to allow `if (sp)` to compile, we must define an automatic conversion to either an arithmetic or a pointer type.

A conversion to arithmetic type is not recommended, as the earlier experience with `operator bool` witnesses. A pointer is not an arithmetic type, period. A conversion to a pointer type makes a lot more sense, and here the problem branches.

If you want to provide automatic conversions to the pointee type (see previous section), then you have two choices: Either you risk unattended calls to operator `delete`, or you

forgo the `if (sp)` test. The tiebreaker is between the lack of convenience and a risky life. The winner is safety, so you cannot write `if (sp)`. Instead, you can choose between `if(sp != 0)` and the more baroque `if(!!sp)`. End of story.

If you don't want to provide automatic conversions to the pointee type, there is an interesting trick you can use to make `if (sp)` possible. Inside the `SmartPtr` class template, define an inner class `Tester` and define a conversion to `Tester*`, as shown in the following code:

```
template <class T>
class SmartPtr
{
   class Tester
   {
      void operator delete(void*);
   };
public:
   operator Tester*() const
   {
      if (!pointee_) return 0;
      static Tester test;
      return &test;
   }
   ...
};
```

Now if you write `if (sp)`, `operator Tester*` enters into action. This operator returns a null value if and only if `pointee_` is null. `Tester` itself disables operator `delete`, so if somebody calls `delete sp`, a compile-time error occurs. Interestingly, `Tester`'s definition itself lies in the private part of `SmartPtr`, so the client code cannot do anything else with it.

`SmartPtr` addresses the issue of tests for equality and inequality as follows:

- Define `operator==` and `operator!=` in two flavors (templated and nontemplated).
- Define `operator!`.
- If you allow automatic conversion to the pointee type, then define an additional conversion to `void*` to ambiguate a call to the `delete` operator intentionally; otherwise, define a private inner class `Tester` that declares a private `operator delete`, and define a conversion to `Tester*` for `SmartPtr` that returns a null pointer if and only if the `pointee_` is null.

7.9 Ordering Comparisons

The ordering comparison operators are `operator<`, `operator<=`, `operator>`, and `operator>=`. You can implement them all in terms of `operator<`.

Whether to allow ordering of smart pointers is an interesting question in and of itself and relates to the dual nature of pointers, which consistently confuses programmers. Pointers are two concepts in one: iterators and monikers. The iterative nature of pointers allows you to walk through an array of objects using a pointer. Pointer arithmetic, including com-

parisons, supports this iterative nature of pointers. At the same time, pointers are monikers—inexpensive object representatives that can travel quickly and access the objects in a snap. The dereferencing operators * and -> support the moniker concept.

The two natures of pointers can be confusing at times, especially when you need only one of them. For operating with a vector, you might use both iteration and dereferencing, whereas for walking through a linked list or for manipulating individual objects, you use only dereferencing.

Ordering comparisons for pointers is defined only when the pointers belong to the same contiguous memory. In other words, you can use ordering comparisons only for pointers that point to elements in the same array.

Defining ordering comparisons for smart pointers boils down to this question: Do smart pointers to the objects in the same array make sense? On the face of it, the answer is no. Smart pointers' main feature is to manage object ownership, and objects with separate ownership do not usually belong to the same array. Therefore, it would be dangerous to allow users to make nonsensical comparisons.

If you really need ordering comparisons, you can always use explicit access to the raw pointer. The issue here is, again, to find the safest and most expressive behavior for most situations.

The previous section concludes that an implicit conversion to a raw pointer type is optional. If SmartPtr's client chooses to allow implicit conversion, the following code compiles:

```
SmartPtr<Something> sp1, sp2;
if (sp1 < sp2) // Converts sp1 and sp2 to raw pointer type,
               // then performs the comparison
...
```

This means that if we want to disable ordering comparisons, we must be proactive, disabling them explicitly. A way of doing this is to declare them and never define them, which means that any use will trigger a link-time error.

```
template <class T>
class SmartPtr
{ ... };

template <class T, class U>
bool operator<(const SmartPtr<T>&, const U&); // Not defined
template <class T, class U>
bool operator<(const T&, const SmartPtr<U>&); // Not defined
```

However, it is wiser to define all other operators in terms of operator<, as opposed to leaving them undefined. This way, if SmartPtr's users think it's best to introduce smart pointer ordering, they need only define operator<.

```
// Ambiguity buster
template <class T, class U>
bool operator<(const SmartPtr<T>& lhs, const SmartPtr<U>& rhs)
{
   return lhs < GetImpl(rhs);
```

```
}
// All other operators
template <class T, class U>
bool operator>(SmartPtr<T>& lhs, const U& rhs)
{
   return rhs < lhs;
}
... similarly for the other operators ...
```

Note the presence, again, of an ambiguity buster. Now if some library user thinks that `SmartPtr<Widget>` should be ordered, the following code is the ticket:

```
inline bool operator<(const SmartPtr<Widget>& lhs,
   const Widget* rhs)
{
   return GetImpl(lhs) < rhs;
}

inline bool operator<(const Widget* lhs,
   const SmartPtr<Widget>& rhs)
{
   return lhs < GetImpl(rhs);
}
```

It's a pity that the user must define two operators instead of one, but it's so much better than defining eight.

This would conclude the issue of ordering, were it not for an interesting detail. Sometimes it is very useful to have an ordering of arbitrarily located objects, not just objects belonging to the same array. For example, you might need to store supplementary per-object information, and you need to access that information quickly. A map ordered by the address of objects is very effective for such a task.

Standard C++ helps in implementing such designs. Although pointer comparison for arbitrarily located objects is undefined, the standard guarantees that `std::less` yields meaningful results for any two pointers of the same type. Because the standard associative containers use `std::less` as the default ordering relationship, you can safely use maps that have pointers as keys.

`SmartPtr` should support this idiom, too; therefore, `SmartPtr` specializes `std::less`. The specialization simply forwards the call to `std::less` for regular pointers:

```
namespace std
{
   template <class T>
   struct less<SmartPtr<T> >
      : public binary_function<SmartPtr<T>, SmartPtr<T>, bool>
   {
      bool operator()(const SmartPtr<T>& lhs,
         const SmartPtr<T>& rhs) const
      {
         return less<T*>()(GetImpl(lhs), GetImpl(rhs));
      }
   };
}
```

In summary, SmartPtr does not define ordering operators by default. It declares—without implementing—two generic operator<s and implements all other ordering operators in terms of operator<. The user can define either specialized or generic versions of operator<.

SmartPtr specializes std::less to provide an ordering of arbitrary smart pointer objects.

7.10 Checking and Error Reporting

Applications need various degrees of safety from smart pointers. Some programs are computation-intensive and must be optimized for speed, whereas some others (actually, most) are input/output intensive, which allows better runtime checking without degrading performance.

Most often, right inside an application, you might need both models: low safety/high speed in some critical areas, and high safety/lower speed elsewhere.

We can divide checking issues with smart pointers into two categories: initialization checking and checking before dereference.

7.10.1 Initialization Checking

Should a smart pointer accept the null (zero) value?

It is easy to implement a guarantee that a smart pointer cannot be null, and it may be very useful in practice. It means that any smart pointer is always valid (unless you fiddle with the raw pointer by using GetImplRef). The implementation is easy with the help of a constructor that throws an exception if passed a null pointer.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(T* p) : pointee_(p)
   {
      if (!p) throw NullPointerException();
   }
   ...
};
```

On the other hand, the null value is a convenient “not a valid pointer” placeholder and can often be useful.

Whether to allow null values affects the default constructor, too. If the smart pointer doesn’t allow null values, then how would the default constructor initialize the raw pointer? The default constructor could be lacking, but that would make smart pointers harder to deal with. For example, what should you do when you have a SmartPtr member variable but don’t have an appropriate initializer for it at construction time? In conclusion, customizing initialization involves providing an appropriate default value.

7.10.2 Checking Before Dereference

Checking before dereference is important because dereferencing the null pointer engenders undefined behavior. For many applications, undefined behavior is not acceptable, so checking the pointer for validity before dereference is the way to go. Checks before dereference belong to `SmartPtr`'s `operator->` and unary `operator*`.

In contrast to the initialization check, the check before dereference can become a major performance bottleneck in your application, because typical applications use (dereference) smart pointers much more often than they create smart pointer objects. Therefore, you should keep a balance between safety and speed. A good rule of thumb is to start with rigorously checked pointers and remove checks from selected smart pointers as profiling demonstrates a need for it.

Can initialization checking and checking before dereference be conceptually separated? No, because there are links between them. If you enforce strict checking upon initialization, then checking before dereference becomes redundant because the pointer is always valid.

This little efficiency victory brought by encapsulation is, however, amended by the loophole provided by `GetImplRef` which allows arbitrary replacement of the underlying raw pointer.

7.10.3 Error Reporting

The only sensible choice for reporting an error is to throw an exception.

You can do something in the sense of avoiding errors. For example, if a pointer is null upon dereference, you can initialize it on the fly. This is a valid and valuable strategy called *lazy initialization*—you construct the value only when you first need it.

If you want to check things only during debugging, you can use the standard `assert` or similar, more sophisticated macros. The compiler ignores the tests in release mode, so, assuming you remove all null pointer errors during debugging, you reap the advantages of both checking and speed.

`SmartPtr` migrates checking to a dedicated Checking policy. This policy implements checking functions (which can optionally provide lazy initialization) and the error reporting strategy.

7.11 Smart Pointers to `const` and `const` Smart Pointers

Raw pointers allow two kinds of `const`ness: the `const`ness of the pointed-to object and that of the pointer itself. The following is an illustration of these two attributes:

```
const Something* pc = new Something; // points to const object
pc->ConstMemberFunction(); // ok
pc->NonConstMemberFunction(); // error
delete pc; // ok (surprisingly)4
Something* const cp = new Something; // const pointer
cp->NonConstMemberFunction(); // ok
cp = new Something; // error, can't assign to const pointer
const Something* const cpc = new Something; // const, points to const
cpc->ConstMemberFunction(); // ok
```

[4] Every once in a while, the question "Why can you apply the `delete` operator to pointers to `const`?" starts a fierce debate in the comp.std.c++ newsgroup. The fact is, for better or worse, the language allows it.

```
cpc->NonConstMemberFunction(); // error
cpc = new Something; // error, can't assign to const pointer
```

The corresponding uses of `SmartPtr` look like this:

```
// Smart pointer to const object
SmartPtr<const Something> spc(new Something);
// const smart pointer
const SmartPtr<Something> scp(new Something);
// const smart pointer to const object
const SmartPtr<const Something> scpc(new Something);
```

The `SmartPtr` class template can detect the `const`ness of the pointed-to object either through partial specialization or by using the `TypeTraits` template defined in Chapter 2. The latter method is preferable because it does not incur source-code duplication as partial specialization does.

`SmartPtr` imitates the semantics of pointers to `const` objects, `const` pointers, and the combinations thereof.

7.12 Arrays

In most cases, instead of dealing with heap-allocated arrays and using `new[]` and `delete[]`, you're better off with `std::vector`. The standard-provided `std::vector` class template provides everything that dynamically allocated arrays provide, plus much more. The extra overhead incurred is negligible in most cases.

However, "most cases" is not "always." There are many situations in which you don't need and don't want a full-fledged vector; a dynamically allocated array is exactly what you need. It is awkward in these cases to be unable to exploit smart pointer capabilities. There is a certain gap between the sophisticated `std::vector` and dynamically allocated arrays. Smart pointers could close that gap by providing array semantics if the user needs them.

From the viewpoint of a smart pointer to an array, the only important issue is to call `delete[] pointee_` in its destructor instead of `delete pointee_`. This issue is already tackled by the Ownership policy.

A secondary issue is providing indexed access, by overloading `operator[]` for smart pointers. This is technically feasible; in fact, a preliminary version of `SmartPtr` did provide a separate policy for optional array semantics. However, only in very rare cases do smart pointers point to arrays. In those cases, there already is a way of providing indexed accessing if you use `GetImpl`:

```
SmartPtr<Widget> sp = ...;
// Access the sixth element pointed to by sp
Widget& obj = GetImpl(sp)[5];
```

It seems like a bad decision to strive to provide extra syntactic convenience at the expense of introducing a new policy.

`SmartPtr` supports customized destruction via the Ownership policy. You can therefore

arrange array-specific destruction via `delete[]`. However, `SmartPtr` does not provide pointer arithmetic.

7.13 Smart Pointers and Multithreading

Most often, smart pointers help with sharing objects. Multithreading issues affect object sharing. Therefore, multithreading issues affect smart pointers.

The interaction between smart pointers and multithreading takes place at two levels. One is the pointee object level, and the other is the bookkeeping data level.

7.13.1 Multithreading at the Pointee Object Level

If multiple threads access the same object and if you access that object through a smart pointer, it can be desirable to lock the object during a function call made through `operator->`. This is possible by having the smart pointer return a proxy object instead of a raw pointer. The proxy object's constructor locks the pointee object, and its destructor unlocks it. The technique is illustrated in Stroustrup (2000). Some code that illustrates this approach is provided here.

First, let's consider a class `Widget` that has two locking primitives: `Lock` and `Unlock`. After a call to `Lock`, you can access the object safely. Any other threads calling `Lock` will block. When you call `Unlock`, you let other threads lock the object.

```
class Widget
{
   ...
   void Lock();
   void Unlock();
};
```

Next, we define a class template `LockingProxy`. Its role is to lock an object (using the `Lock`/`Unlock` convention) for the duration of `LockingProxy`'s lifetime.

```
template <class T>
class LockingProxy
{
public:
   LockingProxy(T* pObj) : pointee_ (pObj)
   { pointee_->Lock(); }
   ~LockingProxy()
   { pointee_->Unlock(); }
   T* operator->() const
   { return pointee_; }
private:
   LockingProxy& operator=(const LockingProxy&);
   T* pointee_;
};
```

In addition to the constructor and destructor, `LockingProxy` defines an `operator->` that returns a pointer to the pointee object.

Although `LockingProxy` looks somewhat like a smart pointer, there is one more layer to it—the `SmartPtr` class template itself.

```
template <class T>
class SmartPtr
{
   ...
   LockingProxy<T> operator->() const
   { return LockingProxy<T>(pointee_); }
private:
   T* pointee_;
};
```

Recall from Section 7.3, which explains the mechanics of `operator->`, that the compiler can apply `operator->` multiple times to one `->` expression, until it reaches a native pointer. Now imagine you issue the following call (assuming `Widget` defines a function `DoSomething`):

```
SmartPtr<Widget> sp = ...;
sp->DoSomething();
```

Here's the trick: `SmartPtr`'s `operator->` returns a temporary `LockingProxy<T>` object. The compiler keeps applying `operator->`. `LockingProxy<T>`'s `operator->` returns a `Widget*`. The compiler uses this pointer to `Widget` to issue the call to `DoSomething`. During the call, the temporary object `LockingProxy<T>` is alive and locks the object, which means that the object is safely locked. As soon as the call to `DoSomething` returns, the temporary `LockingProxy<T>` object is destroyed, so the `Widget` object is unlocked.

Automatic locking is a good application of smart pointer layering. You can layer smart pointers this way by changing the Storage policy.

7.13.2 Multithreading at the Bookkeeping Data Level

Sometimes smart pointers manipulate data in addition to the pointee object. As you read in Section 7.5, reference-counted smart pointers share some data—namely the reference count—under the covers. If you copy a reference-counted smart pointer from one thread to another, you end up having two smart pointers pointing to the same reference counter. Of course, they also point to the same pointee object, but that's accessible to the user, who can lock it. In contrast, the reference count is not accessible to the user, so managing it is entirely the responsibility of the smart pointer.

Not only reference-counted pointers are exposed to multithreading-related dangers. Reference-linked smart pointers (Section 7.5.4) internally hold pointers to each other, which are shared data as well. Reference linking leads to communities of smart pointers, not all of which necessarily belong to the same thread. Therefore, every time you copy, assign, and destroy a reference-linked smart pointer, you must issue appropriate locking; otherwise, the doubly linked list might get corrupted.

In conclusion, multithreading issues ultimately affect smart pointers' implementation. Let's see how to address the multithreading issue in reference counting and reference linking.

7.13.2.1 Multithreaded Reference Counting

If you copy a smart pointer between threads, you end up incrementing the reference count from different threads at unpredictable times.

As the appendix explains, incrementing a value is not an atomic operation. For incrementing and decrementing integral values in a multithreaded environment, you must use the type `ThreadingModel<T>::IntType` and the `AtomicIncrement` and `AtomicDecrement` functions.

Here things become a bit tricky. Better said, they become tricky if you want to separate reference counting from threading.

Policy-based class design prescribes that you decompose a class into elementary behavioral elements and confine each of them to a separate template parameter. In an ideal world, `SmartPtr` would specify an Ownership policy and a ThreadingModel policy and would use them both for a correct implementation.

In the case of multithreaded reference counting, however, things are much too tied together. For example, the counter must be of type `ThreadingModel<T>::IntType`. Then, instead of using `operator++` and `operator--`, you must use `AtomicIncrement` and `AtomicDecrement`. Threading and reference counting melt together; it is unjustifiably hard to separate them.

The best thing to do is to incorporate multithreading in the Ownership policy. Then you can have two implementations: `RefCounting` and `RefCountingMT`.

7.13.2.2 Multithreaded Reference Linking

Consider the destructor of a reference-linked smart pointer. It likely looks like this:

```
template <class T>
class SmartPtr
{
public:
   ~SmartPtr()
   {
      if (prev_ == this)
      {
         delete pointee_;
      }
      else
      {
         prev_->next_ = next_;
         next_->prev_ = prev_;
      }
   }
   ...
private:
   T* pointee_;
   SmartPtr* prev_;
```

```
        SmartPtr* next_;
    };
```

The code in the destructor performs a classic doubly linked list deletion. To make implementation simpler and faster, the list is circular—the last node points to the first node. This way we don't have to test `prev_` and `next_` against zero for any smart pointer. A circular list with only one element has `prev_` and `next_` equal to `this`.

If multiple threads destroy smart pointers that are linked to each other, clearly the destructor must be atomic (uninterruptible by other threads). Otherwise, another thread can interrupt the destructor of a `SmartPtr`, for instance, between updating `prev_->next_` and updating `next_->prev_`. That thread will then operate on a corrupt list.

Similar reasoning applies to `SmartPtr`'s copy constructor and the assignment operator. These functions must be atomic because they manipulate the ownership list.

Interestingly enough, we cannot apply object-level locking semantics here. The appendix divides locking strategies into *class-level* and *object-level* strategies. A class-level locking operation locks all objects in a given class during that operation. An object-level locking operation locks only the object that's subject to that operation. The former technique leads to less memory being occupied (only one mutex per class) but is exposed to performance bottlenecks. The latter is heavier (one mutex per object) but might be speedier.

We cannot apply object-level locking to smart pointers because an operation manipulates up to three objects: the current object that's being added or removed, the previous object, and the next object in the ownership list.

If we want to introduce object-level locking, the starting observation is that there must be one mutex per pointee object—because there's one list per pointee object. We can dynamically allocate a mutex for each object, but this nullifies the main advantage of reference linking over reference counting. Reference linking was more appealing exactly because it didn't use the free store.

Alternatively, we can use an intrusive approach: The pointee object holds the mutex, and the smart pointer manipulates that mutex. But the existence of a sound, effective alternative—reference-counted smart pointers—removes the incentive to provide this feature.

In summary, smart pointers that use reference counting or reference linking are affected by multithreading issues. Thread-safe reference counting needs integer atomic operations. Thread-safe reference linking needs mutexes. `SmartPtr` provides only thread-safe reference counting.

7.14 Putting It All Together

Not much to go! Here comes the fun part. So far we have treated each issue in isolation. It's now time to collect all the decisions into a unique `SmartPtr` implementation.

The strategy we'll use is the one described in Chapter 1: policy-based class design. Each design aspect that doesn't have a unique solution migrates to a policy. The `SmartPtr` class template accepts each policy as a separate template parameter. `SmartPtr` inherits all these template parameters, allowing the corresponding policies to store state.

Let's recap the previous sections by enumerating the variation points of `SmartPtr`. Each variation point translates into a policy.

- *Storage policy (Section 7.3).* By default, the stored type is `T*` (`T` is the first template parameter of `SmartPtr`), the pointer type is again `T*`, and the reference type is `T&`. The means of destroying the pointee object is the `delete` operator.
- *Ownership policy (Section 7.5).* Popular implementations are deep copy, reference counting, reference linking, and destructive copy. Note that Ownership is not concerned with the mechanics of destruction itself; this is Storage's task. Ownership controls the *moment* of destruction.
- *Conversion policy (Section 7.7).* Some applications need automatic conversion to the underlying raw pointer type; others do not.
- *Checking policy (Section 7.10).* This policy controls whether an initializer for `SmartPtr` is valid and whether a `SmartPtr` is valid for dereferencing.

Other issues are not worth dedicating separate policies to them or have an optimal solution:

- The address-of operator (Section 7.6) is best not overloaded.
- Equality and inequality tests are handled with the tricks shown in Section 7.8.
- Ordering comparisons (Section 7.9) are left unimplemented; however, Loki specializes `std::less` for `SmartPtr` objects. The user may define an `operator<`, and Loki helps by defining all other ordering comparisons in terms of `operator<`.
- Loki defines `const`-correct implementations for the `SmartPtr` object, the pointee object, or both.
- There is no special support for arrays, but one of the canned Storage implementations can dispose of arrays by using `operator delete[]`.

The presentation of the design issues surrounding smart pointers made these issues easier to understand and more manageable because each issue was discussed in isolation. It would be very helpful, then, if the implementation could decompose and treat issues in isolation instead of fighting with all the complexity at once.

Divide et Impera—this old principle coined by Julius Caesar can be of help even today with smart pointers. (I'd bet money he didn't predict that.) We break the problem into small component classes, called *policies*. Each policy class deals with exactly one issue. `SmartPtr` inherits all these classes, thus inheriting all their features. It's that simple—yet incredibly flexible, as you will soon see. Each policy is also a template parameter, which means you can mix and match existing stock policy classes or build your own.

The pointed-to type comes first, followed by each of the policies. Here is the resulting declaration of `SmartPtr`:

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
   template <class> class CheckingPolicy = AssertCheck,
   template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

The order in which the policies appear in `SmartPtr`'s declaration puts the ones that you customize most often at the top.

The following four subsections discuss the requirements of the four policies we have defined. A rule for all policies is that they must have value semantics; that is, they must define a proper copy constructor and assignment operator.

7.14.1 The Storage *Policy*

The Storage policy abstracts the structure of the smart pointer. It provides type definitions and stores the actual `pointee_` object.

If `StorageImpl` is an implementation of the Storage policy and `storageImpl` is an object of type `StorageImpl<T>`, then the constructs in Table 7.1 apply.

Here is the default Storage policy implementation:

```
template <class T>
class DefaultSPStorage
{
protected:
   typedef T* StoredType;    //the type of the pointee_ object
   typedef T* PointerType;   //type returned by operator->
   typedef T& ReferenceType; //type returned by operator*
public:
   DefaultSPStorage() : pointee_(Default())
{}
   DefaultSPStorage(const StoredType& p): pointee_(p) {}
   PointerType operator->() const { return pointee_; }
   ReferenceType operator*() const { return *pointee_; }
   friend inline PointerType GetImpl(const DefaultSPStorage& sp)
   { return sp.pointee_; }
   friend inline const StoredType& GetImplRef(
      const DefaultSPStorage& sp)
   { return sp.pointee_; }
   friend inline StoredType& GetImplRef(DefaultSPStorage& sp)
   { return sp.pointee_; }
protected:
   void Destroy()
   { delete pointee_; }
   static StoredType Default()
   { return 0; }
private:
   StoredType pointee_;
};
```

In addition to `DefaultSPStorage`, other sensible policies to define are:

- `ArrayStorage`, which uses `operator delete[]` inside `Destroy`
- `LockedStorage`, which uses layering to provide a smart pointer that locks data while dereferenced (see Section 7.13.1)
- `HeapStorage`, which uses an explicit destructor call followed by `std::free` to release the data

Table 7.1: Storage Policy Constructs

Expression	*Semantics*
`StorageImpl<T>::StoredType`	The type actually stored by the implementation. Default: `T*`.
`StorageImpl<T>::PointerType`	The pointer type defined by the implementation. This is the type returned by `SmartPtr`'s `operator->`. Default: `T*`. Can be different from `StorageImpl<T>::StoredType` when you're using smart pointer layering (see Sections 7.3 and 7.13.1).
`StorageImpl<T>::ReferenceType`	The reference type. This is the type returned by `SmartPtr`'s `operator*`. Default: `T&`.
`GetImpl(storageImpl)`	Returns an object of type `StorageImpl<T>::StoredType`.
`GetImplRef(storageImpl)`	Returns an object of type `StorageImpl<T>::StoredType&`, qualified with `const` if `storageImpl` is `const`.
`storageImpl.operator->()`	Returns an object of type `StorageImpl<T>::PointerType`. Used by `SmartPtr`'s own `operator->`.
`storageImpl.operator*()`	Returns an object of type `StorageImpl<T>::ReferenceType`. Used by `SmartPtr`'s own `operator*`.
`StorageImpl<T>::StoredType p;` `p = storageImpl.Default();`	Returns the default value (usually zero).
`storageImpl.Destroy()`	Destroys the pointee object.

7.14.2 *The* Ownership *Policy*

The Ownership policy must support intrusive as well as nonintrusive reference counting. Therefore, it uses explicit function calls rather than constructor/destructor techniques, as Koenig (1996) does. The reason is that you can call member functions at any time, whereas constructors and destructors are called automatically and only at specific times.

The Ownership policy implementation takes one template parameter, which is the corresponding pointer type. `SmartPtr` passes `StoragePolicy<T>::PointerType` to `OwnershipPolicy`. Note that `OwnershipPolicy`'s template parameter is a pointer type, not an object type.

If `OwnershipImpl` is an implementation of Ownership and `ownershipImpl` is an object of type `OwnershipImpl<P>`, then the constructs in Table 7.2 apply.

Table 7.2: Ownership Policy Constructs

Expression	*Semantics*
`P val1;` `P val2 = OwnershipImpl.` `Clone(val1);`	Clones an object. It can modify the source value if `OwnershipImpl` uses destructive copy.
`const P val1;` `P val2 = ownershipImpl.` `Clone(val1);`	Clones an object.
`P val;` `bool unique = ownershipImpl.` `Release(val);`	Releases ownership of an object. Returns true if the last reference to the object was released.
`bool dc = OwnershipImpl<P>` `::destructiveCopy;`	States whether `OwnershipImpl` uses destructive copy. If that's the case, `SmartPtr` uses the Colvin/Gibbons trick (Meyers 1999) used in `std::auto_ptr`.

An implementation of Ownership that supports reference counting is shown in the following:

```
template <class P>
class RefCounted
{
   unsigned int* pCount_;
protected:
   RefCounted() : pCount_(new unsigned int(1)) {}
   P Clone(const P & val)
   {
      ++*pCount_;
      return val;
   }
   bool Release(const P&)
   {
      if (!--*pCount_)
      {
         delete pCount_;
         return true;
      }
      return false;
   }
   enum { destructiveCopy = false }; // see below
};
```

Implementing a policy for other schemes of reference counting is very easy. Let's write an Ownership policy implementation for COM objects. COM objects have two functions: AddRef and Release. Upon the last Release call, the object destroys itself. You need only direct Clone to AddRef and Release to COM's Release:

```
template <class P>
class COMRefCounted
{
public:
   static P Clone(const P& val)
   {
      val->AddRef();
      return val;
   }
   static bool Release(const P& val)
   {
      val->Release();
      return false;
   }
   enum { destructiveCopy = false }; // see below
};
```

Loki defines the following Ownership implementations:

- DeepCopy, described in Section 7.5.1. DeepCopy assumes that pointee class implements a member function Clone.
- RefCounted, described in Section 7.5.3 and in this section.
- RefCountedMT, a multithreaded version of RefCounted.
- COMRefCounted, a variant of intrusive reference counting described in this section.
- RefLinked, described in Section 7.5.4.
- DestructiveCopy, described in Section 7.5.5.
- NoCopy, which does not define Clone, thus disabling any form of copying.

7.14.3 *The* Conversion *Policy*

Conversion is a simple policy: It defines a Boolean compile-time constant that says whether or not SmartPtr allows implicit conversion to the underlying pointer type.

If ConversionImpl is an implementation of Conversion, then the construct in Table 7.3 applies.

The underlying pointer type of SmartPtr is dictated by its Storage policy and is StorageImpl<T>::StorageType.

As you would expect, Loki defines precisely two Conversion implementations:

- AllowConversion
- DisallowConversion

Table 7.3: Conversion Policy Construct

Expression	*Semantics*
`bool allowConv = ConversionImpl<P>::allow;`	If `allow` is `true`, `SmartPtr` allows implicit conversion to its underlying pointer type.

Table 7.4: Checking Policy Constructs

Expression	*Semantics*
`S value; checkingImpl.OnDefault(value);`	`SmartPtr` calls `OnDefault` in the default constructor call. If `CheckingImpl` does not define this function, it disables the default constructor at compile time.
`S value; checkingImpl.OnInit(value);`	`SmartPtr` calls `OnInit` upon a constructor call.
`S value; checkingImpl.OnDereference (value);`	`SmartPtr` calls `OnDereference` before returning from `operator->` and `operator*`.
`const S value; checkingImpl.OnDereference (value);`	`SmartPtr` calls `OnDereference` before returning from the `const` versions of `operator->` and `operator*`.

7.14.4 *The* Checking *Policy*

As discussed in Section 7.10, there are two main places to check a `SmartPtr` object for consistency: during initialization and before dereference. The checks themselves might use `assert`, exceptions, or lazy initialization or not do anything at all.

The Checking policy operates on the `StoredType` of the Storage policy, not on the `PointerType`. (See Section 7.14.1 for the definition of Storage.)

If `S` is the stored type as defined by the Storage policy implementation, and if `CheckingImpl` is an implementation of Checking, and if `checkingImpl` is an object of type `CheckingImpl<S>`, then the constructs in Table 7.4 apply.

Loki defines the following implementations of Checking:

- `AssertCheck`, which uses `assert` for checking the value before dereferencing.
- `AssertCheckStrict`, which uses `assert` for checking the value upon initialization.
- `RejectNullStatic`, which does not define `OnDefault`. Consequently, any use of `SmartPtr`'s default constructor yields a compile-time error.

- `RejectNull`, which throws an exception if you try to dereference a null pointer.
- `RejectNullStrict`, which does not accept null pointers as initializers (again, by throwing an exception).
- `NoCheck`, which handles errors in the grand C and C++ tradition—that is, it does no checking at all.

7.15 Summary

Congratulations! You have just read one of the longest, wildest chapters of this book—an effort that we hope has paid off. Now you know a lot of things about smart pointers and are equipped with a pretty comprehensive and configurable `SmartPtr` class template.

Smart pointers imitate built-in pointers in syntax and semantics. In addition, they perform a host of tasks that built-in pointers cannot. These tasks might include ownership management and checking against invalid values.

Smart pointer concepts go beyond actual pointer behavior; they can be generalized into smart resources, such as monikers (handles that don't have pointer syntax, yet resemble pointer behavior in the way they enable resource access).

Because they nicely automate things that are very hard to manage by hand, smart pointers are an essential ingredient of successful, robust applications. As small as they are, they can make the difference between a successful project and a failure—or, more often, between a correct program and one that leaks resources like a sieve.

That's why a smart pointer implementer should invest as much attention and effort in this task as possible; the investment is likely to pay in the long term. Similarly, smart pointer users should understand the conventions that smart pointers establish and use them in accordance with those conventions.

The presented implementation of smart pointers focuses on decomposing the areas of functionality into independent policies that the main class template `SmartPtr` mixes and matches. This is possible because each policy implements a well-defined interface.

7.16 `SmartPtr` Quick Facts

- `SmartPtr` declaration:

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
   template <class> class CheckingPolicy = AssertCheck,
   template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

- `T` is the type to which `SmartPtr` points. `T` can be a primitive type or a user-defined type. The `void` type is allowed.
- For the remaining class template parameters (`OwnershipPolicy`, `ConversionPolicy`, `CheckingPolicy`, and `StoragePolicy`), you can implement your own policies or choose from the defaults mentioned in Sections 7.14.1 through 7.14.4.

- OwnershipPolicy controls the ownership management strategy. You can select from the predefined classes DeepCopy, RefCounted, RefCountedMT, COMRefCounted, RefLinked, DestructiveCopy, and NoCopy, described in Section 7.14.2.
- ConversionPolicy controls whether implicit conversion to the pointee type is allowed. The default is to forbid implicit conversion. Either way, you can still access the pointee object by calling GetImpl. You can use the AllowConversion and DisallowConversion implementations (Section 7.14.3).
- CheckingPolicy defines the error checking strategy. The defaults provided are AssertCheck, AssertCheckStrict, RejectNullStatic, RejectNull, RejectNullStrict, and NoCheck (Section 7.14.4).
- StoragePolicy defines the details of how the pointee object is stored and accessed. The default is DefaultSPStorage, which, when instantiated with a type T, defines the reference type as T&, the stored type as T*, and the type returned from operator-> as T* again. Other storage types defined by Loki are ArrayStorage, LockedStorage, and HeapStorage (Section 7.14.1).

This page intentionally left blank

Index

A

Abstract Factory design pattern, 69, 40–51
  architectural role of, 219–222
  basic description of, 219–234
  implementing, 226–233
  interface, generic, 223–226
  quick facts, 233–234
AbstractEnemyFactory, 221–222, 224–227
AbstractFactory, 219–234
AbstractProduct, 209–214, 216–217
abstract products, 209–214, 216–217, 222, 231
Accept, 240–251, 254
AcceptImpl, 252, 256, 259
ACE, 309
Acquire, 161–162, 305, 306
Action, 100
Adapter, 284
Add, 278–282, 284, 300
address-of, 170–171
after, 16
AbstactFactoryUnit, 223–226
Alexandrescu, Andrei, 29
algorithms
  copy, 40
  compile-time, 76
  linear search, 56
  operating on typelists, 76
Allocate, 83, 85
Allocated, 82
allocation, small-object. *See also* allocators
  basic description of, 77–96
  default free store allocators and, 78
  fixed-size allocators and, 84–87
  hat trick and, 89–91
  memory chunks and, 81–84
  quick facts, 94–95
allocators. *See also* allocation, small-object
  default free store, 78
  fixed-size, 84–87
  memory chunks and, 81–84
  workings of, 78–79
allocChunk_, 85, 87
AllowConversion, 192–193
ALU (arithmetic logic unit), 303
AnotherDerived, 197
APIs (application program interfaces), 161, 306
Append, 57–58, 76
Application, 100
arguments, 114–115, 285–290
Array policy, 19–20
arrays, 19–20, 183–184
ArrayStorage, 189–190
assert, 193
AssertCheck, 193
AssertCheckStrict, 193
assertions, compile-time, 23–26
associative collections, 203
AssocVector, 210, 277–278
asynchronous execution, 302
atexit, 134–139, 142–143, 149
ATEXIT_FIXED, 139
AtExitFn, 144–145
AtomicAdd, 305
AtomicAssign, 305
AtomicDecrement, 186
AtomicIncrement, 186
auto_ptr, 108, 159, 170
available_, 79

B

backEnd_, 281, 294
BackendType, 294
BadMonster, 219–222
BadSoldier, 219–222, 230
BadSuperMonster, 219–221
BankAccount, 306
Bar, 139
Base, 63, 90, 91, 176
BASE_IF, 37
BaseLhs, 272, 299
BaseProductList, 227
BaseRhs, 272, 284, 299
BaseSmartPtr, 46
BaseVisitable, 249, 251, 254
BaseVisitor, 245, 249–250, 252, 254, 256, 261
BaseVisitorImpl, 260, 262
BasicDispatcher, 277–284, 292–294, 297–299
BasicFastDispatcher, 291–294, 297, 298–299
before, 16
BinderFirst, 121
BindFirst, 121, 127–128
binding, 119–121, 127–128
BitBlast, 40–41, 44–46
blocksAvailable_, 81–83
blockSize_, 82
bookkeeping data level, 185–187
bool, 105, 174–175, 177, 191
_buffer, 135
bulk allocation, 86
butterfly allocation, 86
Button, 50, 61–62, 219–221

C

callable entities, 103–104
callbackMap_, 280
callbacks, 103–104, 205, 280–281
callbacks_, 205, 281
CallbackType, 277, 280, 299
CastingPolicy policy, 288, 299
CatchAll policy, 260, 262
Chain, 122, 128
chaining requests, 122
char, 35, 38, 114–115
checking issues, 181–182
checkingImpl, 193
Checking policy, 15–16, 188, 193–194
chunks, of memory, 81–87
chunkSize, 88
Circle, 203
class(es). *See also* inheritance; policy classes
  base, 228
  client, 153–154
  decomposing, 19–20
  derived, 228
  final, 29
  generating, with typelists, 64–75
  -level locking operations, 187
  local, 28–29
  object factories and, 200–201
  visitable, 248–255
  visitor, 248–255
Class, 200
ClassLevelLockable, 153, 307–309
Clone, 30, 107, 123, 164, 211–213, 230, 232, 234, 284
CloneFactory, 214–218
clone object factories, 211–215
CLOS, 263
columns_, 292
COM (Component Object Model), 133, 192
command(s). *See also* Command design pattern
  active, 102
  forwarding, 102
Command, 100
Command design pattern, 99–104
  basic description of, 100–102
  in the real world, 102–103
comparison operators, 178–181
compile-time
  assertions, 23–26
  detecting convertibility and inheritance at, 34–37
CompileTimeChecker, 25–26
CompileTimeError, 25
COMRefCounted, 192
ConcreteCommand, 100, 102
ConcreteFact, 232, 234
ConcreteFactory, 226–228, 230–231, 233–234
ConcreteLifetimeTracker, 144–145
concrete products, 222, 227
const, 44, 114–115, 182–183
constant(s)
  mapping integral, to types, 29–31
  -time multimethods, 290–293
ConventionalDialog, 219–221
conversion
  argument, 114–115, 285–290
  binding as, 119–121
  implicit, to raw pointer types, 171–173
  return type, 114–114
  user-defined, 172

Conversion policy, 36–37, 188, 192–193
convertibility, detecting, at compile time, 35–37
copy
construction, eliding of, 123
deep, 123, 162–164, 192
destructive, 168–170
on write (COW), 165
Copy, 40, 45
copy_backward, 144
copyAlgo, 45
CORBA (Common Object Request Broker Architecture), 115, 133
counting, reference, 165–167
covariant return types, 212–213
Create, 9, 11–12, 14, 31–32, 198, 224, 234
CreateButton, 50
CreateDocument, 199
CreateObject, 208–209
CreateScrollBar, 50
CreateShape, 205–206
CreateShapeCallback, 205
CreateStatic, 153
CreateT, 223
CreateUsingMalloc, 153
CreateUsingNew, 153
CreateWindow, 50
Creation policy, 151, 153
Creator policy, 7–9, 11–14, 149, 154, 156
cyclic
dependencies, 243–248
references, 168
CyclicVisitor, 255–257, 261
Czarnecki, Krzysztof, 54

D

dead reference problem, 135–142
Deallocate, 82–87
deallocChunk_, 86–87
deep copy, 123, 163–164, 192
DeepCopy, 192
DEFAULT_CHUNK_SIZE, 93–94, 95
default free store allocator, 78
DefaultLifetime, 153
DEFAULT_THREADING, 94
#define preprocessor directive, 93, 139
DEFINE_CYCLIC_VISITABLE, 257
DEFINE_VISITABLE, 252, 254, 256
delete, 12, 89–91, 94, 108, 132, 143, 159, 172–178
delete[], 184, 189–190
DeleteChar, 125
dependency, circular, 141
DependencyManager, 141
Deposit, 306
dereference, checking before, 182
Derived, 90, 197–198
DerivedToFront algorithm, 63–65, 76, 272
design patterns
Abstract Factory pattern, 69, 49–51, 219–234
Command pattern, 99–104
Double-Checked Locking pattern, 146–147, 149
Prototype design pattern, 228–233
Strategy design pattern, 8
Destroy policy, 20
destroyed_, 136, 138
_DestroySingleton, 135
DestructiveCopy, 192
destructors, 12–13
detection, dead-reference, 135–137
Dialog, 219–221
Dijkstra, Edgar, 305–306
DisallowConversion, 192–193
DispatcherBackend policy, 294
DispatchRhs, 269–270
Display, 135–142, 169
DisplayStatistics, 230
do-it-all interface, failure of, 4–5
DocElement, 236–248, 249, 251, 256, 259
DocElementVisitor, 239–248
DocElementVisitor.h, 243–244
DoClone, 213
DoCreate, 223–224, 227, 232
DocStats, 236–237, 240–242, 246–247, 248
Document, 198
DocumentManager, 198–199
DottedLine, 212
double, 306
Double-Checked Locking pattern, 146–147, 149
DoubleDispatch, 267, 268
double dispatcher, logarithmic, 263, 276–285, 297–300
double switch-on-type, 264–268
Dr. Dobb's Journal, 125
Drawing, 201–203
DrawingDevices, 290
DrawingType, 203
Dylan, 263
dynamic_cast, 238, 243, 246, 251, 255, 267, 285–290, 299
cost of, 255–256
DynamicCaster, 288, 289

E

EasyLevelEnemyFactory, 227, 229, 231
Effective C++ (Meyers), 132
efficient resource use, 302
Eisenecker, Ulrich, 54
ElementAt, 20
Ellipse, 203, 271, 273
else, 238
EmptyType, 39–40, 48, 110
encapsulation, 99
EnforceNotNull, 15, 18
enum, 45
equality, 173–178
erase, 278
Erase, 58–59, 76
EraseAll, 76, 59
error(s)
  messages, compile-time assertions and, 24
  reporting, 181–182
EventHandler, 71
events, 71, 309
exceptions, 209
Execute, 100, 102
Executor, 269–272
exists2Way, 36
ExtendedWidget, 18, 164

F

factor(ies)
  basic description of, 197–218
  classes and, 200–201
  generalization and, 207–210
  implementing, 201–206
  need for, 198–200
  quick facts, 216–217
  templates, 216–218
  type identifiers and, 206–207
  using, with generic components, 215
Factory, 207–208, 215–217
FactoryErrorImpl, 209
FactoryError policy, 208–209, 217–218, 234
FactoryErrorPolicy, 217–218, 234
FastWidgetPtr, 17
Field, 67–70, 74–75
Fire, 275, 270, 271
firstAvailableBlock_, 81–83
FixedAllocator, 80–81, 84–95
FnDispatcher, 280–282, 285, 288, 293–294, 299–300
FnDoubleDispatcher, 280
Foo, 103, 139
forwarding functions, cost of, 122–124
free, 143, 189–190
fun_, 113, 115
functionality, optional, through incomplete instantiation, 13–14
functions
  forwarding, cost of, 122–124
  static, singletons and, 130
functor(s)
  argument type conversions and, 114–115
  basic description of, 99–128
  binding and, 119–121
  chaining requests and, 122
  command design pattern and, 100–102
  double dispatch to, 282–285
  generalized, 99–128
  handling, 110–112
  heap allocation and, 124–125
  implementing Undo and Redo with, 125–126
  multimethods and, 282–285
  quick facts, 126–128
  real-world issues and, 122–125
  return type conversions and, 114–115
Functor1, 106
Functor2, 106
FunctorDispatcher, 283–285, 288, 293, 299–300
FunctorHandler, 110–112, 117–119, 126
Functor template, 99–108, 114, 117, 120, 125–126, 215
FunctorImpl, 107–112, 123–124, 128, 283, 284
FunctorType, 284
FunkyDialog, 219–221

G

GameApp, 230
Gamma, Ralph, 248
generalization, 207–210. *See also* generalized functors
generalized functors. *See also* functors
  argument type conversions and, 114–115
  basic description of, 99–128
  binding and, 119–121
  chaining requests and, 122
  Command design pattern and, 100–102
  handling, 110–112
  heap allocation and, 124–125
  implementing Undo and Redo with, 125–126

quick facts, 126–128
real-world issues and, 122–125
return type conversions and, 114–115
GenLinearHierarchy, 71–75, 227, 230, 231
GenScatterHierarchy, 64–75, 223–226, 227–228
geronimosWork, 117
GetClassIndex, 292–293
GetClassIndexStatic, 292
GetImpl, 162, 173, 183, 190
GetImplRef, 162, 190
GetLongevity, 154
GetPrototype, 12, 14
Go, 269, 270, 272, 279
GoF (Gang of Four) book, 100, 122, 125, 199, 235, 248–249, 255–262
granular interfaces, 224
GraphicButton, 61–62
GUI (graphical user interface), 102

H

handles, 161
Harrison, Tim, 146
Haskell, 263
hat trick, 89–91
HatchingDispatcher, 272
HatchingExecutor, 271
HatchRectanglePoly, 279
header files
DocElementVisitor.h, 243–244
Multimethods.h, 294
Typelist.h, 51, 55, 75
SmallAlloc.h, 93–95
heaps, 124–125, 189–190
HeapStorage, 189–190
hierarchies
linear, 70–74
scattered, 64–70
HTMLDocument, 198

I

IdentifierType, 209, 213, 215
IdToProductMap, 214
#ifdef preprocessor directive, 138–139
if-else statements, 29, 267, 268, 270
if statements, 238, 267, 305
IMPLEMENT_INDEXABLE_CLASS, 293
implicit conversion, to raw pointer types, 171–173
IncrementFontSize, 240, 241
indexed access, 55
IndexOf, 56, 76, 275
inequality, 173–178
inheritance
detecting, at compile time, 34–37
logarithmic dispatcher and, 279
multiple, 5–6
INHERITS, 37
InIt, 40, 82
initialization
checking, 181–182
dynamic, 132
lazy, 182
object factories and, 197
static, 132
insert, 205
InsertChar, 120, 122, 125–126
Instance, 131–132, 135–137, 146, 151
instantiation, 13–14, 120, 272, 274
int, 159
Int2Type, 29–31, 68–69
interface(s)
Abstract Factory design pattern, 223–226
application program (APIs), 161, 306
do-it-all, failure of, 4–5
granular, 224
graphical user (GUI), 102
separation, 101
intrusive reference counting, 167. *See also* reference counting
IntType, 186, 304
InvocationTraits, 275
isConst, 47
isPointer, 47
isReference, 41, 47
isStdArith, 47
isStdFloat, 47
isStdFundamental, 42–43, 47
isStdIntegral, 47
isStdSignedInt, 47
isStdUnsignedInt, 47
isVolatile, 47

K

KDL problem, 135–142, 155
Keyboard, 135–142, 155
KillPhoenixSingleton, 138
Knuth, Donald E., 77, 78

L

Lattanzi, Len, 77
Length, 76
less, 180–181
LhsTypes, 272
Lifetime policy, 149–153
LifetimeTracker, 142–143
LIFO (last in, first out), 134
Line, 203, 204, 212–213
linking, reference, 167–168
LISP, 52
ListOfTypes, 295
Lock, 146, 184, 306
LockedStorage, 189–190
locking
  class-level, 307
  object-level, 306–308
  pattern, double-checked, 146–147, 149
  semantics, 306–308
LockingProxy, 184–185
Log, 135–142
logarithmic double dispatcher, 263, 276–285, 297–300
logic_error, 152
Loki, 70, 77, 91–92, 210, 303, 309
  multimethods and, 263, 268, 277–278, 288, 295–300
  mutexes and, 306
  smart pointers and, 163
long double data type, 35
longevity, 139–145, 149, 151
lower_bound, 277

M

MacroCommand, 122
macros, 122, 251–252, 254, 256–257, 261
"maelstrom effect," 170
MAKE_VISITABLE, 261
MakeAdapter, 28–29
MakeCopy, 164
MakeT, 223
malloc, 143
map, 204–205, 210, 277, 278
mapping
  integral constants to types, 31–32
  type-to-type, 31–33
Martin, Robert, 245
maxObjectSize, 88
MAX_SMALL_OBJECT_SIZE, 93–94, 95
MemControlBlock, 78–79
MemFunHandler, 117–119, 126
memory
  allocators, workings of, 78–80
  chunks of, 81–87
  heaps, 124–125, 189–190
  RMW (read-modify-write) operation, 303–304
Meyers, Scott, 77, 133–134, 276, 280
Meyers singleton, 133–134
ML, 263
ModalDialog, 27
Monster, 219–222, 224, 229
More Effective C++ (Meyers), 276, 280
MostDerived, 63, 76
multimethods
  arguments and, 285–290
  basic description of, 263–300
  constant-time, 290–293
  double switch-on-type and, 265–268
  logarithmic double dispatcher and, 276–285
  need for, 264–265
  quick facts, 297–300
  symmetry and, 273–74
Multimethods.h, 294
MultiThreaded, 6
MultiThreadedRefCounting, 186–187
multithreading, 145–148, 302–306
  at the bookkeeping data level, 185–187
  critique of, 302–303
  library, 301–309
  mutexes and, 305–306
  at the pointee object level, 184–185
  reference counting and, 186
  reference tracking and, 186–187
  smart pointers and, 184–187
mutex_, 146
mutexes, 146–147, 305–306
MyController, 27
MyOnlyPrinter, 130
MyVisitor, 257

N

name, 206
name cyclic dependency, 243
new[], 183
next_, 187
NiftyContainer, 28–30, 33–34
NoChecking, 15, 18–19
NoCopy, 192
NoDestroy, 153
NoDuplicates, 60–61, 76

NonConstType, 47
nontemplated operators, 176–177
NonVolatileType, 47
NoQualifiedType, 47
NullType, 39–41, 48, 52, 54–56, 62–63, 270

O

object factor(ies)
  basic description of, 197–218
  classes and, 200–201
  generalization and, 207–210
  implementing, 201–206
  need for, 198–200
  quick facts, 216–217
  templates, 216–218
  type identifiers and, 206–207
  using, with generic components, 215
ObjectLevelLockable, 307–309
OnDeadReference, 136–139, 150
OnError, 272
OnEvent, 70–71
OnUnknownVisitor, 260, 262
operators
  operator.*, 104, 116–117
  operator!=, 174, 176, 178
  operator(), 103–113, 116, 122–127, 283, 285
  operator->, 157, 160–162, 165, 182–185
  operator->*, 104, 116–117
  operator*, 157, 161–162, 178–179, 182
  operator<, 144
  operator<=, 178–179
  operator=, 162
  operator==, 176–177, 178
  operator>, 178–179
  operator>=, 178–179
  operator[], 55, 183, 278
  operator T*, 172
OpNewCreator, 10
OpNewFactoryUnit, 226, 227, 231, 234
OrderedTypeInfo, 217, 276–277
orthogonal policies, 20
OutIt, 40
ownership-handling strategies, 163–170
Ownership policy, 183–184, 186, 188, 190–192

P

Paragraph, 237, 241, 247, 249, 254, 256
ParagraphVisitor, 246, 247, 248, 251
parameter(s)
  template, 10–11, 64, 105
  types, optimized, 43–44
ParameterType, 43–44, 47, 123
ParentFunctor, 111, 120–121
Parm1, 111
Parm2, 111
ParmN, 109
Parrot, 117–119
pattern(s)
  Abstract Factory pattern, 69, 49–51, 219–234
  Command pattern, 99–104
  Double-Checked Locking pattern, 146–147, 149
  Prototype design pattern, 228–233
  Strategy design pattern, 8
Pattern Hatching (Vlissides), 133
pData_, 86
pDocElem, 246
pDuplicateShape, 212
pDynObject, 140
pFactory_, 222
Phoenix Singleton, 137–142, 149, 153
pimpl idiom, 78
pInstance_, 131–132, 136, 138, 146–148, 151
placement new, 138
pLastAlloc_, 89
POD (plain old data) structure, 45–46
Point3D, 70
pointee_, 160, 161, 178, 183
PointeeType, 41–42, 47, 160–161
pointee object level, 184–185
pointer(s)
  address-of operator and, 170–171
  arrays and, 183–184
  basic description of, 157–195
  checking issues and, 181–182
  copy on write (COW) and, 165
  deep copy and, 163–164
  destructive copy and, 168–170
  equality and, 173–178
  error reporting and, 181–182
  failure of the do-it-all interface and, 5
  handling, to member functions, 115–119
  implicit conversion and, 171–173
  inequality and, 173–178
  multithreading and, 184–187
  ordering comparison operators and, 178–181
  ownership-handling strategies, 163–170
  quick facts, 194–195
  raw, 171–173
  reference counting and, 165–167, 186
  reference linking and, 166–168
  reference tracking and, 186–187

traits of, implementing, 41–42
types, implicit conversion, 171–173
PointerToObj, 118
PointerTraits, 41–42
PointerType, 160, 190–191
policies. *See also* policy classes
basic description of, 3, 7–11
BasicDispatcher and, 293–294
BasicFastDispatcher and, 293–294
compatible, 17–18
decomposing classes in, 19–20
enriched, 12
multimethods and, 293–294
noncompatible, 17–18
orthogonal, 20
singletons and, 149–150, 152–153
stock, 152–153
policies (listed by name). *See also* policies
Array policy, 19–20
CastingPolicy policy, 288, 299
CatchAll Policy, 260, 262
Checking policy, 15–16, 188, 193–194
Conversion policy, 36–37, 188, 192–193
Creation policy, 151, 153
Creator policy, 8–9, 11–14, 149, 155, 156
Destroy policy, 20
DispatcherBackend policy, 294
FactoryError policy, 208–209, 217–218, 234
Lifetime policy, 149–153
Ownership policy, 183–184, 186, 188, 190–192
Storage policy, 17, 185, 188, 189–190
Structure policy, 16–17
ThreadingModel policy, 16, 149, 151–153, 186, 303–309
policy classes. *See also* classes
basic description of, 3–22
combining, 14–16
customizing structure with, 16–17
destructors of, 12–13
implementing, 10–11
Poly, 271, 279
Polygon, 203, 212
polymorphism, 78, 163–164
Abstract Factory implementation and, 228–229
multimethods and, 264
object factories and, 197–200, 211–212
prev_, 187
Printer, 161–162
printf, 105
printingPort_, 130
priority_queue, 142–145
ProductCreator, 210–211, 216–217
Prototype design pattern, 228–233
PrototypeFactoryUnit, 231–232, 234
prototypes, 228–233
pTrackerArray, 143

Q

qualifiers, stripping, 44

R

RasterBitmap, 237, 241, 247, 249, 256
RasterBitmapVisitor, 247, 251
realloc, 143
Receiver, 100
Rectangle, 267, 279, 289
RectanglePoly, 279
Redo, 125–126
RefCounted, 6, 192
RefCountedMT, 192
reference(s)
counting, 165–167, 186
linking, 167–168, 186–187
ReferencedType, 41–42, 47
RefLinked, 192
RegisterShape, 206
RejectNull, 194
RejectNullStatic, 193–194
RejectNullStrict, 194
Release, 161–162, 192, 305, 306
Replace, 76
ReplaceAll, 61, 76
Reset, 162
resource leaks, 133
Result, 55
ResultType, 111, 269, 284
return type(s)
conversion, 114–115
covariant, 228
generalized functors and, 114–115
RISC processors, 148
RMW (read-modify-write) operation, 303–304
RoundedRectangle, 272, 286–287, 289
RoundedShape, 286–287, 289
runtime_error, 137, 209–210, 300
runtime type information (RTTI), 215, 245, 255, 276

S

safe_reinterpret_cast, 26
SafeWidgetPtr, 17
sameType, 36
Save, 201–203
scanf, 105
ScheduleDestruction, 149–150
Schmidt, Douglas, 146–147
Scroll, 122
ScrollBar, 50, 61–62
Secretary, 5
Section, 254
Select, 33–34, 62
semantics
  failure of the do-it-all interface and, 4–5
  functors and, 99
  locking, 306–308
semaphores, 309
SetLongevity, 140–145, 149
SetPrototype, 12, 14, 232
Shape, 201–202, 265, 268, 279, 286–289
ShapeCast, 289
ShapeFactory, 204–205, 215
Shapes, 290
SillyMonster, 219–222, 226
SillySoldier, 219–222, 230
SillySuperMonster, 219–222
SingleThreaded, 153, 308
singleton(s). *See also* SingletonHolder
  basic C++ idioms supporting, 131–132
  dead reference problem and, 135–142
  destroying, 133–135
  Double-Checked Locking pattern and, 146–147
  failure of the do-it-all interface and, 4–5
  implementing, 129–154
  longevity and, 139–145
  Meyers singleton, 133–134
  multithreading and, 145–148
  Phoenix, 137–142, 149, 153
  static functions and, 130
  uniqueness of, enforcing, 132–133
SingletonHolder, 3, 91, 129–130, 148–153, 215. *See also* singletons
  assembling, 150–152
  decomposing, 149–150
  quick facts, 155–156
  working with, 153–156
SingletonWithLongevity, 153, 154
sizeof, 25, 34–35, 82, 90–91
size_t, 36, 79
skinnable programs, 103
SmallAlloc.h, 93–95
small-object allocation
  basic description of, 77–96
  default free store allocators and, 78
  fixed-size allocators and, 84–87
  hat trick and, 89–91
  memory chunks and, 81–84
  quick facts, 94–95
SmallObjAllocator, 80–81, 87–89, 92–95
SmallObject, 80–81, 89, 91–95, 123–124
smart pointer(s)
  address-of operator and, 170–171
  arrays and, 183–184
  basic description of, 157–195
  checking issues and, 181–182
  copy on write (COW) and, 165
  deep copy and, 163–164
  destructive copy and, 168–170
  equality and, 173–178
  error reporting and, 181–182
  failure of the do-it-all interface and, 5
  implicit conversion and, 171–173
  inequality and, 173–178
  multithreading and, 184–187
  ordering comparison operators and, 178–181
  ownership-handling strategies, 163–170
  quick facts, 194–195
  raw, 171–173
  reference counting and, 165–167, 186
  reference linking and, 167–168
  reference tracking and, 186–187
  types, implicit conversion, 171–173
SmartPtr, 3, 6–7, 14–19, 44, 87, 157–195
Soldier, 219–222, 224, 229–230
SomeLhs, 278, 284
SomeRhs, 278, 284
SomeThreadingModel, 309
SomeVisitor, 250, 254
spImpl_, 112
statements
  if, 238, 267, 305
  if-else, 29, 267, 268, 270
static, 280
static_cast, 114, 285–290, 299
STATIC_CHECK, 25
StaticDispatcher, 268–276, 297–298
static manipulation, 6
Statistics, 249
stats, 246–247
STL, 115, 171
StorageImpl, 189, 190
Storage policy, 17, 185, 188, 189–190
Strategy design pattern, 8

String, 302–303, 307
Stroustrup, Bjarne, 202
struct, 45
structure(s)
  customizing, with policy classes, 16–17
  POD (plain old data), 45–46
  specialization of, 7
Structure policy, 16–17
SuperMonster, 219–222, 229
SUPERSUBCLASS, 37, 62
Surprises.cpp, 224
Sutter, Herb, 77
switch, 203
SwitchPrototype, 13–14
symmetry, 273–74
synchronization objects, 303

T

template(s)
  advantages of, 6–7
  implementing policy classes with, 11
  skeleton, Functor class, 104–108
  specialization, partial, 53–54
  template parameters, 10–11, 64, 105
templated operators, 176–177
TemporarySecretary, 5
Tester, 178
Tester*, 178
TestFunction, 113–115
TextDocument, 198
ThreadingModel policy, 16, 149, 151–153, 186, 303–309
time
  separation, 101
  slicing, 201
TList, 53–65, 75, 120, 127, 223, 231–234, 261
Tools menu, 102
trampoline functions (thunks), 280–283
Trampoline, 281
translation units, 132
Triangle, 289
tuples, generating, 70
type(s)
  atomic operations on, 303–304
  conversions, generalized functors and, 114–115
  detection of fundamental, 42–43
  identifiers, 201, 206–207
  integral, 303–305
  modifiers, 308–309
  multiple inheritance and, 6
  multithreading and, 303–305
  parameters, optimized, 43–44
  replacing an element in, 60–61
  safety, loss of static, 4
  selection, 33–34
  -to-type mapping, 31–33
  traits, 38–46
Type2Type, 32–33, 223
TypeAt, 55, 76
TypeAtNonStrict, 55, 76, 109
typedef, 14, 19, 51, 54, 215, 295
typeid, 38, 267
type_info, 37–39, 54, 206–207, 213–215, 276–277, 295
TypeInfo, 38–39, 48
TYPELIST, 295
Typelist.h, 51, 55, 75
typelists, 223, 268, 272
  appending to, 57–58
  basic description of, 49–76
  calculating length and, 53–54
  class generation with, 64–75
  compile-time algorithms operating on, 76
  creating, 52–53
  defining, 51–52
  detecting fundamental types and, 42–43
  need for, 49–41
  partially ordering, 61–64
  quick facts, 75
  searching, 56–57
TypesLhs, 269–270, 274
TypesRhs, 269, 270, 274
TypeTraits, 41–48, 123, 183

U

UCHAR_MAX, 83
Undo, 125–126
unique bouncing virtual functions, 238
Unlock, 184
UpdateStats, 237–238
upper_bound, 144
use_Size, 91

V

ValueType, 33
vector, 183
VectorGraphic, 244

VectorizedDrawing, 259
Veldhuizen, Todd, 54
virtual
  constructor dilemma, 229
  functions, 238
Visit, 249–250, 254, 256
Visitable, 249, 251, 252–254
Visitor design pattern
  acyclic, 243–248
  basic description of, 235–262
  catch-all function and, 242–243, 258–259
  cyclic, 243–248, 254–257
  generic implementation of, 248–255
  hooking variations, 258–260
  nonstrict visitation and, 261
  quick facts, 261–262
VisitParagraph, 240, 242
VisitRasterBitmap, 242
VisitVectorGraphic, 244
Vlissides, John, 133
void*, 172
volatile, 151, 308–309, 308–309
VolatileType, 151, 309

W

Widget, 26–32, 38, 44, 61–62, 159, 164, 184, 309
WidgetEventHandler, 71–74
WidgetFactory, 49–51
WidgetInfo, 65–67
WidgetManager, 9–14, 19–20
Window, 50
Withdrawal, 306

X

X Windows, 103