¡¶Effective C++¡·
&
¡¶More Effective C++¡·

FMD KBS ¸¨ÖúÖÆ×÷
http://www.fmdstudio.net

Effective C++, Second Edition

Contents

Dedication

For Nancy, without whom nothing would be much worth doing.

Continue to Preface

Back to Dedication
Continue to Acknowledgments

Preface

This book is a direct outgrowth of my experiences teaching C++ to professional programmers. I've found that most students, after a week of intensive instruction, feel comfortable with the basic constructs of the language, but they tend to be less sanguine about their ability to put the constructs together in an effective manner. Thus began my attempt to formulate short, specific, easy-to-remember guidelines for effective software development in C++: a summary of the things experienced C++ programmers almost always do or almost always avoid doing.

I was originally interested in rules that could be enforced by some kind of `lint`-like program. To that end, I led research into the development of tools to examine C++ source code for violations of user-specified conditions.[1] Unfortunately, the research ended before a complete prototype could be developed. Fortunately, several commercial C++-checking products are now available. (You'll find an overview of such products in the article on static analysis tools by me and Martin Klaus.)

Though my initial interest was in programming rules that could be automatically enforced, I soon realized the limitations of that approach. The majority of guidelines used by good C++ programmers are too difficult to formalize or have too many important exceptions to be blindly enforced by a program. I was thus led to the notion of something less precise than a computer program, but still more focused and to-the-point than a general C++ textbook. The result you now hold in your hands: a book containing 50 specific suggestions on how to improve your C++ programs and designs.

In this book, you'll find advice on what you should do, and why, and what you should not do, and why not. Fundamentally, of course, the whys are more important than the whats, but it's a lot more convenient to refer to a list of guidelines than to memorize a textbook or two.

Unlike most books on C++, my presentation here is not organized around particular language features. That is, I don't talk about constructors in one place, about virtual functions in another, about inheritance in a third, etc. Instead, each discussion in the book is tailored to the guideline it accompanies, and my coverage of the various aspects of a particular language feature may be dispersed throughout the book.

The advantage of this approach is that it better reflects the complexity of the software systems for which C++ is often chosen, systems in which understanding individual language features is not enough. For example, experienced C++ developers know that understanding inline functions and understanding virtual destructors does not necessarily mean you understand inline virtual destructors. Such battle-scarred developers recognize that comprehending the *interactions* between the features in C++ is of the greatest possible importance in using the language effectively. The

organization of this book reflects that fundamental truth.

The disadvantage of this design is that you may have to look in more than one place to find everything I have to say about a particular C++ construct. To minimize the inconvenience of this approach, I have sprinkled cross-references liberally throughout the text, and a comprehensive index is provided at the end of the book.

In preparing this second edition, my ambition to improve the book has been tempered by fear. Tens of thousands of programmers embraced the first edition of *Effective C++*, and I didn't want to destroy whatever characteristics attracted them to it. However, in the six years since I wrote the book, C++ has changed, the C++ library has changed (see Item 49), my understanding of C++ has changed, and accepted usage of C++ has changed. That's a lot of change, and it was important to me that the technical material in *Effective C++* be revised to reflect those changes. I'd done what I could by updating individual pages between printings, but books and software are frighteningly similar — there comes a time when localized enhancements fail to suffice, and the only recourse is a system-wide rewrite. This book is the result of that rewrite: *Effective C++*, Version 2.0.

Those familiar with the first edition may be interested to know that every Item in the book has been reworked. I believe the overall structure of the book remains sound, however, so little there has changed. Of the 50 original Items, I retained 48, though I tinkered with the wording of a few Item titles (in addition to revising the accompanying discussions). The retired Items (i.e., those replaced with completely new material) are numbers 32 and 49, though much of the information that used to be in Item 32 somehow found its way into the revamped Item 1. I swapped the order of Items 41 and 42, because that made it easier to present the revised material they contain. Finally, I reversed the direction of my inheritance arrows. They now follow the almost-universal convention of pointing from derived classes to base classes. This is the same convention I followed in my 1996 book, *More Effective C++*.

The set of guidelines in this book is far from exhaustive, but coming up with good rules — ones that are applicable to almost all applications almost all the time — is harder than it looks. Perhaps you know of additional guidelines, of more ways in which to program effectively in C++. If so, I would be delighted to hear about them.

On the other hand, you may feel that some of the Items in this book are inappropriate as general advice; that there is a better way to accomplish a task examined in the book; or that one or more of the technical discussions is unclear, incomplete, or misleading. I encourage you to let me know about these things, too.

◦Donald Knuth has a long history of offering a small reward to people who notify him of errors in his books. The quest for a perfect book is laudable in any case, but in view of the number of bug-ridden C++ books that have been rushed to market, I feel especially strongly compelled to follow Knuth's example. Therefore, for each error in this book that is reported to me — be it technical,

grammatical, typographical, or otherwise — I will, in future printings, gladly add to the acknowledgments the name of the first person to bring that error to my attention.

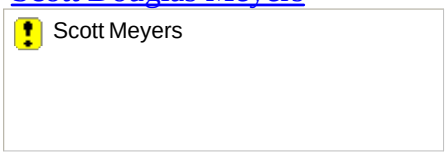
Send your suggested guidelines, your comments, your criticisms, and — sigh — your bug reports to:

Scott Meyers
c/o Publisher, Corporate and Professional Publishing
Addison Wesley Longman, Inc.
1 Jacob Way
Reading, MA 01867
U. S. A.

Alternatively, you may send electronic mail to ec++@awl.com.

I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. This list is available at the ◦*Effective C++* World Wide Web site. If you would like a copy of this list, but you lack access to the World Wide Web, please send a request to one of the addresses above, and I will see that the list is sent to you.

◦Scott Douglas Meyers



Stafford, Oregon
July 1997

Back to Dedication
Continue to Acknowledgments

---

[1] You can find an overview of the research at the ◦*Effective C++* World Wide Web site.
Return

Back to Preface
Continue to Introduction

Acknowledgments

Some three decades have elapsed since Kathy Reed taught me what a computer was and how to program one, so I suppose this is really all her fault. In 1989, Donald French asked me to develop C++ training materials for the Institute for Advanced Professional Studies, so perhaps he should shoulder some blame. The students in my class at Stratus Computer the week of June 3, 1991, were not the first to suggest I write a book summarizing the pearls of alleged wisdom that tumble forth when I teach, but they were the ones who finally convinced me to do it, so they bear some of the responsibility. I'm grateful to them all.

Many of the Items and examples in this book have no particular source, at least not one I can remember. Instead, they grew out of a combination of my own experiences using and teaching C++, those of my colleagues, and opinions expressed by contributors to the Usenet C++ newsgroups. Many examples that are now standard in the C++ teaching community — notably strings — can be traced back to the initial edition of Bjarne Stroustrup's *The C++ Programming Language* (Addison-Wesley, 1986). Several of the Items found here (e.g., Item 17) can also be found in that seminal work.

Item 8 includes an implementation idea from Steve Clamage's May 1993 *C++ Report* article, "Implementing `new` and `delete`." Item 9 was motivated by commentary in *The Annotated C++ Reference Manual* (see Item 50), and Items 10 and 13 were suggested by John Shewchuk. The implementation of `operator new` in Item 10 is based on presentations in the second edition of Stroustrup's *The C++ Programming Language* (Addison-Wesley, 1991) and Jim Coplien's *Advanced C++: Programming Styles and Idioms* (Addison-Wesley, 1992). Dietmar Kü hl pointed out the undefined behavior I describe in Item 14. Doug Lea provided the aliasing examples at the end of Item 17. The idea of using `0L` for `NULL` in Item 25 came from Jack Reeves's March 1996 *C++ Report* article, "Coping with Exceptions." Several members of various Usenet C++ newsgroups helped refine that Item's class for implementing `NULL`-based pointer conversions via member templates. A newsgroup posting by Steve Clamage tempered my enthusiasm for references to functions in Item 28. Item 33 incorporates observations from Tom Cargill's *C++ Programming Style* (Addison-Wesley, 1992), Martin Carroll's and Margaret Ellis's *Designing and Coding Reusable C++* (Addison-Wesley, 1995), *Taligent's Guide to Designing Programs* (Addison-Wesley, 1994), Rob Murray's *C++ Strategies and Tactics* (Addison-Wesley, 1993), as well as information from publications and newsgroup postings by Steve Clamage. The material in Item 34 benefited from my discussions with John Lakos and from reading his book, *Large-Scale C++ Software Design* (Addison-Wesley, 1996). The envelope/letter terminology in that Item comes from Jim Coplien's *Advanced C++: Programming Styles and Idioms*; John Carolan coined the delightful term, "Cheshire Cat class." The rectangle/square example of Item 35 is taken from Robert Martin's March 1996 *C++ Report* column, "The Liskov Substitution Principle." A long-ago `comp.lang.c++` posting by Mark Linton set me straight in my thinking about grasshoppers and crickets in Item 43.

My traits examples in Item 49 are taken from Nathan Myers's June 1995 ◦*C++ Report* article, ◦"A New and Useful Template Technique: Traits," and Pete Becker's "C/C++ Q&A" column in the November 1996 ◦*C/C++ User's Journal*; my summary of C++'s internationalization support is based on a pre-publication book draft by Angelika Langer and Klaus Kreft. Of course, "Hello world" comes from ◦*The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, first published in 1978).

Many readers of the first edition sent suggestions I was unable to incorporate in that version of the book, but that I've adopted in one form or another for this new edition. Others took advantage of Usenet C++ newsgroups to post insightful remarks about the material in the book. I'm grateful to each of the following individuals, and I've noted where I took advantage of their ideas: Mike Kaelbling and Julio Kuplinsky (Introduction); a person my notes identify only as "a guy at Claris"[2] (Item 5); Joel Regen and Chris Treichel (Item 7); Tom Cargill, Larry Gajdos, Doug Morgan, and Uwe Steinmü ller (Item 10); Roger Scott and Steve Burkett (Item 12); David Papurt (Item 13); Alexander Gootman (Item 14); David Bern (Item 16); Tom Cargill, Tom Chappell, Dan Franklin, and Jerry Liebelson (Item 17); John "Eljay" Love-Jensen (Item 19); Eric Nagler (Item 22); Roger Eastman, Doug Moore, and Aaron Naiman (Item 23); Dat Thuc Nguyen (Item 25); Tony Hansen, Natraj Kini, and Roger Scott (Item 33); John Harrington, Read Fleming, and Dave Smallberg (Item 34); Johan Bengtsson (Item 36); Rene Rodoni (Item 39); Paul Blankenbaker and Mark Somer (Item 40); Tom Cargill and John Lakos (Item 41); Frieder Knauss and Roger Scott (Item 42); David Braunegg, Steve Clamage, and Dawn Koffman (Item 45); Tom Cargill (Item 46); Wesley Munsil (Item 47); Randy Mangoba (most class definitions); and John "Eljay" Love-Jensen (many places where I use `type double`).

Partial and/or complete drafts of the manuscript for the first edition were reviewed by Tom Cargill, Glenn Carroll, Tony Davis, Brian Kernighan, Jak Kirman, Doug Lea, Moises Lejter, Eugene Santos, Jr., John Shewchuk, John Stasko, Bjarne Stroustrup, Barbara Tilly, and Nancy L. Urbano. In addition, I received suggestions for improvements that I was able to incorporate in later printings from the following alert readers, whom I've listed in the order in which I received their reports: Nancy L. Urbano, Chris Treichel, David Corbin, Paul Gibson, Steve Vinoski, Tom Cargill, Neil Rhodes, David Bern, Russ Williams, Robert Brazile, Doug Morgan, Uwe Steinmü ller, Mark Somer, Doug Moore, Dave Smallberg, Seth Meltzer, Oleg Shteynbuk, David Papurt, Tony Hansen, Peter McCluskey, Stefan Kuhlins, David Braunegg, Paul Chisholm, Adam Zell, Clovis Tondo, Mike Kaelbling, Natraj Kini, Lars Nyman, Greg Lutz, Tim Johnson, John Lakos, Roger Scott, Scott Frohman, Alan Rooks, Robert Poor, Eric Nagler, Antoine Trux, Cade Roux, Chandrika Gokul, Randy Mangoba, and Glenn Teitelbaum. Each of these people was instrumental in improving the book you now hold.

Drafts of the second edition were reviewed by Derek Bosch, Tim Johnson, Brian Kernighan, Junichi Kimura, Scott Lewandowski, Laura Michaels, Dave Smallberg, Clovis Tondo, Chris Van Wyk, and Oleg Zabluda. I am grateful to all these people, but especially to Tim Johnson, whose detailed review influenced the final manuscript in dozens of ways. I am also grateful to Jill Huchital and Steve Reiss for their assistance in finding good reviewers, a task of crucial importance

and increasing difficulty. Dawn Koffman and Dave Smallberg suggested improvements to the ◦C++ training materials derived from my books, and many of their ideas have found their way into this revision. Finally, I received comments from the following readers of earlier printings of this book, and I've modified this current printing to take their suggestions into account: Daniel Steinberg, Arunprasad Marathe, Doug Stapp, Robert Hall, Cheryl Ferguson, Gary Bartlett, Michael Tamm, Kendall Beaman, Eric Nagler, Max Hailperin, Joe Gottman, Richard Weeks, Valentin Bonnard, Jun He, Tim King, Don Maier, Ted Hill, Mark Harrison, Michael Rubenstein, Mark Rodgers, David Goh, Brenton Cooper, and Andy Thomas-Cramer.

Evi Nemeth (with the cooperation of ◦Addison-Wesley, the ◦USENIX Association, and ◦The Internet Engineering Task Force) has agreed to see to it that leftover copies of the first edition are delivered to computer science laboratories at universities in Eastern Europe; these universities would otherwise find it difficult to acquire such books. Evi voluntarily performs this service for several authors and publishers, and I'm happy to be able to help in some small way. If you'd like more information on this program, contact Evi.

Sometimes it seems that the players in publishing change nearly as frequently as the trends in programming, so I'm pleased that my editor, John Wait, my marketing director, Kim Dawley, and my production director, Marty Rabinowitz, continue to play the roles they did in those innocent days of 1991 when I first started this whole authoring thing. Sarah Weaver was my project manager for this book, Rosemary Simpson provided advice on indexing, and Lana Langlois acted as my primary contact and all-around bercoordinator at Addison-Wesley until she left for greener — or at least different — pastures. I thank them and their colleagues for helping with the thousand tasks that separate simple writing from actual publishing.

Kathy Wright had nothing to do with the book, but she'd like to be acknowledged.

For the first edition, I am grateful for the enthusiastic and unflagging encouragement provided by my wife, Nancy L. Urbano, and by my family and hers. Although writing a book was the last thing I was supposed to be doing, and doing so reduced my free time from merely little to effectively none, they made it clear that the effort was worth it if, in the end, the result was an author in the family.

That author has been in the family six years now, yet Nancy continues to tolerate my hours, put up with my technochatter, and encourage my writing. She also has a knack for knowing just the right word when I can't think of it. The Nancyless life is not worth living.

Our dog, ◦Persephone, never lets me confuse my priorities. Deadline or no deadline, the time for a walk is always *now*.

Back to Preface
Continue to Introduction

---

[2] Note to this guy: I was at Claris the week of November 15, 1993. Contact me and identify yourself

as the one who pointed out the importance of specifying which form of `delete` to use with a `typedef`, and I'll happily give you proper credit in these acknowledgments. I'll even throw in a little something (*very* little — don't get excited) to help compensate for my pathetic failure to know who you are.
Return

Back to Acknowledgments
Continue to Shifting from C to C++

Introduction

Learning the fundamentals of a programming language is one thing; learning how to design and implement *effective* programs in that language is something else entirely. This is especially true of C++, a language boasting an uncommon range of power and expressiveness. Built atop a full-featured conventional language (C), it also offers a wide range of object-oriented features, as well as support for templates and exceptions.

Properly used, C++ can be a joy to work with. An enormous variety of designs, both object-oriented and conventional, can be expressed directly and implemented efficiently. You can define new data types that are all but indistinguishable from their built-in counterparts, yet are substantially more flexible. A judiciously chosen and carefully crafted set of classes — one that automatically handles memory management, aliasing, initialization and clean-up, type conversions, and all the other conundrums that are the bane of software developers — can make application programming easy, intuitive, efficient, and nearly error-free. It isn't unduly difficult to write effective C++ programs, *if* you know how to do it.

Used without discipline, C++ can lead to code that is incomprehensible, unmaintainable, inextensible, inefficient, and just plain wrong.

The trick is to discover those aspects of C++ that are likely to trip you up and to learn how to avoid them. That is the purpose of this book. I assume you already know C++ as a *language* and that you have some experience in its use. What I provide here is a guide to using the language *effectively*, so that your software is comprehensible, maintainable, extensible, efficient, and likely to behave as you expect.

The advice I proffer falls into two broad categories: general design strategies, and the nuts and bolts of specific language features.

The design discussions concentrate on how to choose between different approaches to accomplishing something in C++. How do you choose between inheritance and templates? Between templates and generic pointers? Between public and private inheritance? Between private inheritance and layering? Between function overloading and parameter defaulting? Between virtual and nonvirtual functions? Between pass-by-value and pass-by-reference? It is important to get these decisions right at the outset, because an incorrect choice may not become apparent until much later in the development process, at which point its rectification is often difficult, time-consuming, demoralizing, and expensive.

Even when you know exactly what you want to do, getting things just right can be tricky. What's the proper return type for the assignment operator? How should `operator new` behave when it can't find enough memory? When should a destructor be virtual? How should you write a member initialization list? It's crucial to sweat details like these, because failure to do so almost always leads to unexpected, possibly mystifying, program behavior. More importantly, the aberrant

behavior may not be immediately apparent, giving rise to the specter of code that passes through quality control while still harboring a variety of undetected bugs — ticking time-bombs just waiting to go off.

This is not a book that must be read cover to cover to make any sense. You need not even read it front to back. The material is broken down into 50 Items, each of which stands more or less on its own. Frequently, however, one Item will refer to others, so one way to read the book is to start with a particular Item of interest and then follow the references to see where they lead you.

The Items are grouped into general topic areas, so if you are interested in discussions related to a particular issue, such as memory management or object-oriented design, you can start with the relevant section and either read straight through or start jumping around from there. You will find, however, that all of the material in this book is pretty fundamental to effective C++ programming, so almost everything is eventually related to everything else in one way or another.

This is not a reference book for C++, nor is it a way for you to learn the language from scratch. For example, I'm eager to tell you all about the gotchas in writing your own `operator new` (see Items 7-10), but I assume you can go elsewhere to discover that that function must return a `void*` and its first argument must be of type `size_t`. There are a number of introductory books on C++ that contain information such as that.

The purpose of *this* book is to highlight those aspects of C++ programming that are usually treated superficially (if at all). Other books describe the different parts of the language. This book tells you how to combine those parts so you end up with effective programs. Other books tell you how to get your programs to compile. This book tells you how to avoid problems that compilers won't tell you about.

Like most languages, C++ has a rich folklore that is usually passed from programmer to programmer as part of the language's grand oral tradition. This book is my attempt to record some of that accumulated wisdom in a more accessible form.

At the same time, this book limits itself to legitimate, *portable*, C++. Only language features in the ◦ISO/ANSI language standard (see Item M35) have been used here. In this book, portability is a key concern, so if you're looking for implementation-dependent hacks and kludges, this is not the place to find them.

Alas, C++ as described by the standard is sometimes different from the C++ supported by your friendly neighborhood compiler vendors. As a result, when I point out places where relatively new language features are useful, I also show you how to produce effective software in their absence. After all, it would be foolish to labor in ignorance of what the future is sure to bring, but by the same token, you can't just put your life on hold until the latest, greatest, be-all-and-end-all C++ compilers appear on your computer. You've got to work with the tools available to you, and this book helps you do just that.

Notice that I refer to *compilers* — plural. Different compilers implement varying approximations to the standard, so I encourage you to develop your code under at least two compilers. Doing so will help you avoid inadvertent dependence on one vendor's proprietary language extension or its

misinterpretation of the standard. It will also help keep you away from the bleeding edge of compiler technology, i.e., from new features supported by only one vendor. Such features are often poorly implemented (buggy or slow — frequently both), and upon their introduction, the C++ community lacks experience to advise you in their proper application. Blazing trails can be exciting, but when your goal is producing reliable code, it's often best to let others do the bushwhacking for you.

One thing you will *not* find in this book is the C++ Gospel, the One True Path to perfect C++ software. Each of the 50 Items in this book provides guidance on how to come up with better designs, how to avoid common problems, or how to achieve greater efficiency, but none of the Items is universally applicable. Software design and implementation is a complex task, one invariably colored by the constraints of the hardware, the operating system, and the application, so the best I can do is provide *guidelines* for creating better programs.

If you follow all the guidelines all the time, you are unlikely to fall into the most common traps surrounding C++, but guidelines, by their very nature, have exceptions. That's why each Item has an explanation. The explanations are the most important part of the book. Only by understanding the rationale behind an Item can you reasonably determine whether it applies to the software you are developing and to the unique constraints under which you toil.

The best use of this book, then, is to gain insight into how C++ behaves, why it behaves that way, and how to use its behavior to your advantage. Blind application of the Items in this book is clearly inappropriate, but at the same time, you probably shouldn't violate any of the guidelines without having a good reason for doing so.

There's no point in getting hung up on terminology in a book like this; that form of sport is best left to language lawyers. However, there is a small C++ vocabulary that everybody should understand. The following terms crop up often enough that it is worth making sure we agree on what they mean.

A *declaration* tells compilers about the name and type of an object, function, class, or template, but it omits certain details. These are declarations:

```
extern int x;                           // object declaration

int numDigits(int number);              // function declaration

class Clock;                            // class declaration

template<class T>
class SmartPointer;                     // template declaration
```

A *definition*, on the other hand, provides compilers with the details. For an object, the definition is where compilers allocate memory for the object. For a function or a function template, the definition provides the code body. For a class or a class template, the definition lists the members of the class or template:

```
int x;                                  // object definition

int numDigits(int number)               // function definition
{                                       // (this function returns
  int digitsSoFar = 1;                  // the number of digits in
                                        // its parameter)

  if (number < 0) {
    number = -number;
    ++digitsSoFar;
  }

  while (number /= 10) ++digitsSoFar;

  return digitsSoFar;
}

class Clock {                            // class definition
public:
  Clock();
  ~Clock();

  int hour() const;
  int minute() const;
  int second() const;

  ...

};

template<class T>
class SmartPointer {                     // template definition
public:
  SmartPointer(T *p = 0);
  ~SmartPointer();

  T * operator->() const;
  T& operator*() const;

  ...

};
```

That brings us to constructors. A *default constructor* is one that can be called without any arguments. Such a constructor either has no parameters or has a default value for every parameter. You generally need a default constructor if you want to define arrays of objects:

```
class A {
public:
  A();                                   // default constructor
};

A arrayA[10];                            // 10 constructors called
```

```
class B {
public:
  B(int x = 0);                          // default constructor
};

B arrayB[10];                            // 10 constructors called,
                                         // each with an arg of 0

class C {
public:
  C(int x);                              // not a default constructor
};

C arrayC[10];                            // error!
```

You may find that your compilers reject arrays of objects when a class's default constructor has default parameter values. For example, some compilers refuse to accept the definition of `arrayB` above, even though it receives the blessing of the C++ standard. This is an example of the kind of discrepancy that can exist between the standard's description of C++ and a particular compiler's implementation of the language. Every compiler I know of has a few of these shortcomings. Until compiler vendors catch up to the standard, be prepared to be flexible, and take solace in the certainty that someday in the not-too-distant future, the C++ described in the standard will be the same as the language accepted by C++ compilers.

Incidentally, if you want to create an array of objects for which there is no default constructor, the usual ploy is to define an array of *pointers* instead. Then you can initialize each pointer separately by using `new`:

```
C *ptrArray[10];                         // no constructors called

ptrArray[0] = new C(22);                 // allocate and construct
                                         // 1 C object

ptrArray[1] = new C(4);                  // ditto

...
```

This suffices almost all the time. When it doesn't, you'll probably have to fall back on the more advanced (and hence more obscure) "placement `new`" approach described in Item M4.

Back on the terminology front, a *copy constructor* is used to initialize an object with a different object of the same type:

```
class String {
public:
  String();                              // default constructor
  String(const String& rhs);             // copy constructor
  ...

private:
  char *data;
```

```
};

String s1;                                  // call default constructor
String s2(s1);                              // call copy constructor
String s3 = s2;                             // call copy constructor
```

Probably the most important use of the copy constructor is to define what it means to pass and return objects by value. As an example, consider the following (inefficient) way of writing a function to concatenate two `String` objects:

```
const String operator+(String s1, String s2)
{
  String temp;

  delete [] temp.data;

  temp.data =
    new char[strlen(s1.data) + strlen(s2.data) + 1];

  strcpy(temp.data, s1.data);
  strcat(temp.data, s2.data);

  return temp;
}

String a("Hello");
String b(" world");
String c = a + b;                           // c = String("Hello world")
```

This `operator+` takes two `String` objects as parameters and returns one `String` object as a result. Both the parameters and the result will be passed by value, so there will be one copy constructor called to initialize `s1` with `a`, one to initialize `s2` with `b`, and one to initialize `c` with `temp`. In fact, there might even be some additional calls to the copy constructor if a compiler decides to generate intermediate temporary objects, which it is allowed to do (see Item M19). The important point here is that pass-by-value *means* "call the copy constructor."

By the way, you wouldn't really implement `operator+` for `String`s like this. Returning a `const String` object is correct (see Items 21 and 23), but you would want to pass the two parameters by reference (see Item 22).

Actually, you wouldn't write `operator+` for `String`s at all if you could help it, and you should be able to help it almost all the time. That's because the standard C++ library (see Item 49) contains a string type (cunningly named `string`), as well as an `operator+` for `string` objects that does almost exactly what the `operator+` above does. In this book, I use both `String` and `string` objects, but I use them in different ways. (Note that the former name is capitalized, the latter name is not.) If I need just a generic string and I don't care how it's implemented, I use the `string` type that is part of the standard C++ library. That's what you should do, too. Often, however, I want to make a point about how C++ behaves, and in those cases, I need to show some implementation code. That's when I use the (nonstandard) `String` class. As a programmer, you should use the standard `string`

type whenever you need a string object; the days of developing your own string class as a C++ rite of passage are behind us. However, you still need to understand the issues that go into the development of classes like `string`. `String` is convenient for that purpose (and for that purpose only). As for raw `char*`-based strings, you shouldn't use those antique throw-backs unless you have a *very* good reason. Well-implemented `string` types can now be superior to `char*`s in virtually every way — including efficiency (see Item 49 and Items M29-M30).

The next two terms we need to grapple with are *initialization* and *assignment*. An object's initialization occurs when it is given a value for the very first time. For objects of classes or structs with constructors, initialization is *always* accomplished by calling a constructor. This is quite different from object assignment, which occurs when an object that is already initialized is given a new value:

```
string s1;                              // initialization
string s2("Hello");                     // initialization
string s3 = s2;                         // initialization

s1 = s3;                                // assignment
```

From a purely operational point of view, the difference between initialization and assignment is that the former is performed by a constructor while the latter is performed by `operator=`. In other words, the two processes correspond to different function calls.

The reason for the distinction is that the two kinds of functions must worry about different things. Constructors usually have to check their arguments for validity, whereas most assignment operators can take it for granted that their argument is legitimate (because it has already been constructed). On the other hand, the target of an assignment, unlike an object undergoing construction, may already have resources allocated to it. These resources typically must be released before the new resources can be assigned. Frequently, one of these resources is memory. Before an assignment operator can allocate memory for a new value, it must first deallocate the memory that was allocated for the old value.

Here is how a `String` constructor and assignment operator could be implemented:

```
// a possible String constructor
String::String(const char *value)
{
  if (value) {                          // if value ptr isn't null
    data = new char[strlen(value) + 1];
    strcpy(data,value);
  }
  else {                                // handle null value ptr[3]

    data = new char[1];
    *data = '\0';                       // add trailing null char
  }
}

// a possible String assignment operator
String& String::operator=(const String& rhs)
```

```
{
  if (this == &rhs)
    return *this;                          // see Item 17

  delete [] data;                          // delete old memory
  data =                                   // allocate new memory
    new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;                            // see Item 15
}
```

Notice how the constructor must check its parameter for validity and how it must take pains to ensure that the member `data` is properly initialized, i.e., points to a `char*` that is properly null-terminated. On the other hand, the assignment operator takes it for granted that its parameter is legitimate. Instead, it concentrates on detecting pathological conditions, such as assignment to itself (see Item 17), and on deallocating old memory before allocating new memory. The differences between these two functions typify the differences between object initialization and object assignment. By the way, if the "`[]`" notation in the use of `delete` is new to you (pardon the pun), Items 5 and M8 should dispel any confusion you may have.

A final term that warrants discussion is *client*. A client is a programmer, one who uses the code you write. When I talk about clients in this book, I am referring to people looking at your code, trying to figure out what it does; to people reading your class definitions, attempting to determine whether they want to inherit from your classes; to people examining your design decisions, hoping to glean insights into their rationale.

You may not be used to thinking about your clients, but I'll spend a good deal of time trying to convince you to make their lives as easy as you can. After all, you are a client of the software other people develop. Wouldn't you want those people to make things easy for you? Besides, someday you may find yourself in the uncomfortable position of having to use your *own* code, in which case your client will be you!

I use two constructs in this book that may not be familiar to you. Both are relatively recent additions to C++. The first is the `bool` type, which has as its values the keywords `true` and `false`. This is the type now returned by the built-in relational operators (e.g., <, >, ==, etc.) and tested in the condition part of `if`, `for`, `while`, and `do` statements. If your compilers haven't implemented `bool`, an easy way to approximate it is to use a typedef for `bool` and constant objects for `true` and `false`:

```
typedef int bool;

const bool false = 0;
const bool true = 1;
```

This is compatible with the traditional semantics of C and C++. The behavior of programs using this approximation won't change when they're ported to `bool`-supporting compilers. For a different way of approximating `bool` — including a discussion of the advantages and disadvantages of each

approach — turn to the Introduction of *More Effective C++*.

The second new construct is really four constructs, the casting forms `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. Conventional C-style casts look like this:

```
(type) expression                          // cast expression to be of
                                           // type type
```

The new casts look like this:

```
static_cast<type>(expression)              // cast expression to be of
                                           // type type
const_cast<type>(expression)

dynamic_cast<type>(expression)      <type>

reinterpret_cast<type>(expression)  <type>
```

These different casting forms serve different purposes:

- `const_cast` is designed to cast away the constness of objects and pointers, a topic I examine in Item 21.
- `dynamic_cast` is used to perform "safe downcasting," a subject we'll explore in Item 39.
- `reinterpret_cast` is engineered for casts that yield implementation-dependent results, e.g., casting between function pointer types. (You're not likely to need `reinterpret_cast` very often. I don't use it at all in this book.)
- `static_cast` is sort of the catch-all cast. It's what you use when none of the other casts is appropriate. It's the closest in meaning to the conventional C-style casts.

Conventional casts continue to be legal, but the new casting forms are preferable. They're much easier to identify in code (both for humans and for tools like `grep`), and the more narrowly specified purpose of each casting form makes it possible for compilers to diagnose usage errors. For example, only `const_cast` can be used to cast away the constness of something. If you try to cast away an object's or a pointer's constness using one of the other new casts, your cast expression won't compile.

For more information on the new casts, see Item M2 or consult a recent introductory textbook on C++.

In the code examples in this book, I have tried to select meaningful names for objects, classes, functions, etc. Many books, when choosing identifiers, embrace the time-honored adage that brevity is the soul of wit, but I'm not as interested in being witty as I am in being clear. I have therefore striven to break the tradition of using cryptic identifiers in books on programming languages. Nonetheless, I have at times succumbed to the temptation to use two of my favorite parameter names, and their meanings may not be immediately apparent, especially if you've never done time on a compiler-writing chain gang.

The names are `lhs` and `rhs`, and they stand for "left-hand side" and "right-hand side," respectively. I use them as parameter names for functions implementing binary operators, especially `operator==` and arithmetic operators like `operator*`. For example, if `a` and `b` are objects representing rational numbers, and if rational numbers can be multiplied via a non-member `operator*` function, the expression

```
a * b
```

is equivalent to the function call

```
operator*(a, b)
```

As you will discover in Item 23, I declare `operator*` like this:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

As you can see, the left-hand operand, `a`, is known as `lhs` inside the function, and the right-hand operand is known as `rhs`.

I've also chosen to abbreviate names for pointers according to this rule: a pointer to an object of type `T` is often called `pt`, "pointer to T." Here are some examples:

```
string *ps;                             // ps = ptr to string

class Airplane;
Airplane *pa;                           // pa = ptr to Airplane

class BankAccount;
BankAccount *pba;                       // pba = ptr to BankAccount
```

I use a similar convention for references. That is, `rs` might be a reference-to-`string` and `ra` a reference-to-`Airplane`.

I occasionally use the name `mf` when I'm talking about member functions.

On the off chance there might be some confusion, any time I mention the C programming language in this book, I mean the ◦ISO/◦ANSI-sanctified version of C, not the older, less strongly-typed, "classic" C.

Back to Acknowledgments
Continue to Shifting from C to C++

---

[3] My `String`'s constructor taking a `const char*` argument handles the case where a null pointer is passed in, but the standard `string` type is not required to be so tolerant. Attempts to create a `string` from a null pointer yield undefined results. However, it is safe to create a `string` object from an empty `char*`-based string, i.e., from "".
Return

Back to Introduction
Continue to Item 1: Prefer const and inline to #define.

Shifting from C to C++

Getting used to C++ takes a little while for everyone, but for grizzled C programmers, the process can be especially unnerving. Because C is effectively a subset of C++, all the old C tricks continue to work, but many of them are no longer appropriate. To C++ programmers, for example, a pointer to a pointer looks a little funny. Why, we wonder, wasn't a reference to a pointer used instead?

C is a fairly simple language. All it really offers is macros, pointers, structs, arrays, and functions. No matter what the problem is, the solution will always boil down to macros, pointers, structs, arrays, and functions. Not so in C++. The macros, pointers, structs, arrays and functions are still there, of course, but so are private and protected members, function overloading, default parameters, constructors and destructors, user-defined operators, inline functions, references, friends, templates, exceptions, namespaces, and more. The design space is much richer in C++ than it is in C: there are just a lot more options to consider.

When faced with such a variety of choices, many C programmers hunker down and hold tight to what they're used to. For the most part, that's no great sin, but some C habits run contrary to the spirit of C++. Those are the ones that have simply *got* to go.

Back to Introduction
Continue to Item 1: Prefer const and inline to #define.

Back to Shifting from C to C++
Continue to Item 2: Prefer <iostream> to <stdio.h>.

Item 1: Prefer `const` and `inline` to `#define`.

This Item might better be called "prefer the compiler to the preprocessor," because `#define` is often treated as if it's not part of the language *per se*. That's one of its problems. When you do something like this,

```
#define ASPECT_RATIO 1.653
```

the symbolic name `ASPECT_RATIO` may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler. As a result, the name `ASPECT_RATIO` may not get entered into the symbol table. This can be confusing if you get an error during compilation involving the use of the constant, because the error message may refer to `1.653`, not `ASPECT_RATIO`. If `ASPECT_RATIO` was defined in a header file you didn't write, you'd then have no idea where that `1.653` came from, and you'd probably waste time tracking it down. This problem can also crop up in a symbolic debugger, because, again, the name you're programming with may not be in the symbol table.

The solution to this sorry scenario is simple and succinct. Instead of using a preprocessor macro, define a constant:

```
const double ASPECT_RATIO = 1.653;
```

This approach works like a charm. There are two special cases worth mentioning, however.

First, things can get a bit tricky when defining constant pointers. Because constant definitions are typically put in header files (where many different source files will include them), it's important that the *pointer* be declared `const`, usually in addition to what the pointer points to. To define a constant `char*`-based string in a header file, for example, you have to write `const` *twice*:

```
const char * const authorName = "Scott Meyers";
```

For a discussion of the meanings and uses of `const`, especially in conjunction with pointers, see Item 21.

Second, it's often convenient to define class-specific constants, and that calls for a slightly different tack. To limit the scope of a constant to a class, you must make it a member, and to ensure there's

at most one copy of the constant, you must make it a *static* member:

```
class GamePlayer {
private:
  static const int NUM_TURNS = 5;    // constant declaration
  int scores[NUM_TURNS];             // use of constant
  ...
};
```

There's a minor wrinkle, however, which is that what you see above is a *declaration* for `NUM_TURNS`, not a definition. You must still define static class members in an implementation file:

```
const int GamePlayer::NUM_TURNS;      // mandatory definition;
                                      // goes in class impl. file
```

There's no need to lose sleep worrying about this detail. If you forget the definition, your linker should remind you.

Older compilers may not accept this syntax, because it used to be illegal to provide an initial value for a static class member at its point of declaration. Furthermore, in-class initialization is allowed only for integral types (e.g., `int`s, `bool`s, `char`s, etc.), and only for constants. In cases where the above syntax can't be used, you put the initial value at the point of definition:

```
class EngineeringConstants {      // this goes in the class
private:                          // header file


  static const double FUDGE_FACTOR;


  ...


};


// this goes in the class implementation file
```

```
const double EngineeringConstants::FUDGE_FACTOR = 1.35;
```

This is all you need almost all the time. The only exception is when you need the value of a class constant during compilation of the class, such as in the declaration of the array `GamePlayer::scores` above (where compilers insist on knowing the size of the array during compilation). Then the accepted way to compensate for compilers that (incorrectly) forbid the in-class specification of initial values for integral class constants is to use what is affectionately known as "the enum hack." This technique takes advantage of the fact that the values of an enumerated type can be used where `int`s are expected, so `GamePlayer` could just as well have been defined like this:

```
class GamePlayer {
private:
  enum { NUM_TURNS = 5 };    // "the enum hack" — makes
                             // NUM_TURNS a symbolic name
                             // for 5


  int scores[NUM_TURNS];     // fine


...


};
```

Unless you're dealing with compilers of primarily historical interest (i.e., those written before 1995), you shouldn't have to use the enum hack. Still, it's worth knowing what it looks like, because it's not uncommon to encounter it in code dating back to those early, simpler times.

Getting back to the preprocessor, another common (mis)use of the `#define` directive is using it to implement macros that look like functions but that don't incur the overhead of a function call. The canonical example is computing the maximum of two values:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

This little number has so many drawbacks, just thinking about them is painful. You're better off playing in the freeway during rush hour.

Whenever you write a macro like this, you have to remember to parenthesize all the arguments

when you write the macro body; otherwise you can run into trouble when somebody calls the macro with an expression. But even if you get that right, look at the weird things that can happen:

```
int a = 5, b = 0;



max(++a, b);         // a is incremented twice
max(++a, b+10);      // a is incremented once
```

Here, what happens to `a` inside `max` depends on what it is being compared with!

Fortunately, you don't need to put up with this nonsense. You can get all the efficiency of a macro plus all the predictable behavior and type-safety of a regular function by using an inline function (see Item 33):

```
inline int max(int a, int b) { return a > b ? a : b; }
```

Now this isn't quite the same as the macro above, because this version of `max` can only be called with `int`s, but a template fixes that problem quite nicely:

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

This template generates a whole family of functions, each of which takes two objects convertible to the same type and returns a reference to (a constant version of) the greater of the two objects. Because you don't know what the type `T` will be, you pass and return by reference for efficiency (see Item 22).

By the way, before you consider writing templates for commonly useful functions like `max`, check the standard library (see Item 49) to see if they already exist. In the case of `max`, you'll be pleasantly surprised to find that you can rest on others' laurels: `max` is part of the standard C++ library.

Given the availability of `const`s and `inline`s, your need for the preprocessor is reduced, but it's not completely eliminated. The day is far from near when you can abandon `#include`, and `#ifdef`/`#ifndef` continue to play important roles in controlling compilation. It's not yet time to retire the preprocessor, but you should definitely plan to start giving it longer and more frequent vacations.

Back to Shifting from C to C++

Continue to Item 2: Prefer <iostream> to <stdio.h>.

Back to Item 1: Prefer const and inline to #define.
Continue to Item 3: Prefer new and delete to malloc and free.

Item 2: Prefer `<iostream>` to `<stdio.h>`.

Yes, they're portable. Yes, they're efficient. Yes, you already know how to use them. Yes, yes, yes. But venerated though they are, the fact of the matter is that `scanf` and `printf` and all their ilk could use some improvement. In particular, they're not type-safe and they're not extensible. Because type safety and extensibility are cornerstones of the C++ way of life, you might just as well resign yourself to them right now. Besides, the `printf`/`scanf` family of functions separate the variables to be read or written from the formatting information that controls the reads and writes, just like FORTRAN does. It's time to bid the 1950s a fond farewell.

Not surprisingly, these weaknesses of `printf`/`scanf` are the strengths of `operator>>` and `operator<<`.

```
int i;
Rational r;                          // r is a rational number


...


cin >> i >> r;
cout << i << r;
```

If this code is to compile, there must be functions `operator>>` and `operator<<` that can work with an object of type `Rational` (possibly via implicit type conversion — see Item M5). If these functions are missing, it's an error. (The versions for `int`s are standard.) Furthermore, compilers take care of figuring out which versions of the operators to call for different variables, so you needn't worry about specifying that the first object to be read or written is an `int` and the second is a `Rational`.

In addition, objects to be read are passed using the same syntactic form as are those to be written, so you don't have to remember silly rules like you do for `scanf`, where if you don't already have a pointer, you have to be sure to take an address, but if you've already got a pointer, you have to be sure *not* to take an address. Let C++ compilers take care of those details. They have nothing better to do, and you *do* have better things to do. Finally, note that built-in types like `int` are read and written in the same manner as user-defined types like `Rational`. Try *that* using `scanf` and `printf`!

Here's how you might write an output routine for a class representing rational numbers:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);


  ...


private:
  int n, d;    // numerator and denominator


friend ostream& operator<<(ostream& s, const Rational& r);
};


ostream& operator<<(ostream& s, const Rational& r)
{
  s << r.n << '/' << r.d;
  return s;
}
```

This version of `operator<<` demonstrates some subtle (but important) points that are discussed elsewhere in this book. For example, `operator<<` is not a member function (Item 19 explains why), and the `Rational` object to be output is passed into `operator<<` as a reference-to-`const` rather than as an object (see Item 22). The corresponding input function, `operator>>`, would be declared and implemented in a similar manner.

Reluctant though I am to admit it, there are some situations in which it may make sense to fall back on the tried and true. First, some implementations of iostream operations are less efficient than the corresponding C stream operations, so it's possible (though unlikely — see Item M16) that you have an application in which this makes a significant difference. Bear in mind, though, that this says nothing about iostreams *in general*, only about particular implementations; see Item M23. Second, the iostream library was modified in some rather fundamental ways during the course of its standardization (see Item 49), so applications that must be maximally portable may discover that different vendors support different approximations to the standard. Finally, because the classes of the iostream library have constructors and the functions in `<stdio.h>` do not, there are rare occasions involving the initialization order of static objects (see Item 47) when the standard C library may be more useful simply because you know that you can always call it with impunity.

The type safety and extensibility offered by the classes and functions in the iostream library are more useful than you might initially imagine, so don't throw them away just because you're used to `<stdio.h>`. After all, even after the transition, you'll still have your memories.

Incidentally, that's no typo in the Item title; I really mean `<iostream>` and not `<iostream.h>`. Technically speaking, there is no such thing as `<iostream.h>` — the standardization committee eliminated it in favor of `<iostream>` when they truncated the names of the other non-C standard header names. The reasons for their doing this are explained in Item 49, but what you really need to understand is that if (as is likely) your compilers support both `<iostream>` and `<iostream.h>`, the headers are subtly different. In particular, if you `#include <iostream>`, you get the elements of the iostream library ensconced within the namespace `std` (see Item 28), but if you `#include <iostream.h>`, you get those same elements at global scope. Getting them at global scope can lead to name conflicts, precisely the kinds of name conflicts the use of namespaces is designed to prevent. Besides, `<iostream>` is less to type than `<iostream.h>`. For many people, that's reason enough to prefer it.

Back to Item 1: Prefer const and inline to #define.
Continue to Item 3: Prefer new and delete to malloc and free.

Back to Item 2: Prefer <iostream> to <stdio.h>.
Continue to Item 4: Prefer C++-style comments.

Item 3: Prefer `new` and `delete` to `malloc` and `free`.

The problem with `malloc` and `free` (and their variants) is simple: they don't know about constructors and destructors.

Consider the following two ways to get space for an array of 10 `string` objects, one using `malloc`, the other using `new`:

```
string *stringArray1 =
  static_cast<string*>(malloc(10 * sizeof(string)));


string *stringArray2 = new string[10];
```

Here `stringArray1` points to enough memory for 10 `string` objects, but no objects have been constructed in that memory. Furthermore, without jumping through some rather obscure linguistic hoops (such as those described in Items M4 and M8), you have no way to initialize the objects in the array. In other words, `stringArray1` is pretty useless. In contrast, `stringArray2` points to an array of 10 fully constructed `string` objects, each of which can safely be used in any operation taking a `string`.

Nonetheless, let's suppose you magically managed to initialize the objects in the `stringArray1` array. Later on in your program, then, you'd expect to do this:

```
free(stringArray1);



delete [] stringArray2;      // see Item 5 for why the
                             // "[]" is necessary
```

The call to `free` will release the memory pointed to by `stringArray1`, but no destructors will be called on the `string` objects in that memory. If the `string` objects themselves allocated memory, as `string` objects are wont to do, all the memory they allocated will be lost. On the other hand, when `delete` is called on `stringArray2`, a destructor is called for each object in the array before any memory is released.

Because `new` and `delete` interact properly with constructors and destructors, they are clearly the

superior choice.

Mixing `new` and `delete` with `malloc` and `free` is usually a bad idea. When you try to call `free` on a pointer you got from `new` or call `delete` on a pointer you got from `malloc`, the results are undefined, and we all know what "undefined" means: it means it works during development, it works during testing, and it blows up in your most important customers' faces.

The incompatibility of `new`/`delete` and `malloc`/`free` can lead to some interesting complications. For example, the `strdup` function commonly found in `<string.h>` takes a `char*`-based string and returns a copy of it:

```
char * strdup(const char *ps);      // return a copy of what
                                    // ps points to
```

At some sites, both C and C++ use the same version of `strdup`, so the memory allocated inside the function comes from `malloc`. As a result, unwitting C++ programmers calling `strdup` might overlook the fact that they must use `free` on the pointer returned from `strdup`. But wait! To forestall such complications, some sites might decide to rewrite `strdup` for C++ and have this rewritten version call `new` inside the function, thereby mandating that callers later use `delete`. As you can imagine, this can lead to some pretty nightmarish portability problems as code is shuttled back and forth between sites with different forms of `strdup`.

Still, C++ programmers are as interested in code reuse as C programmers, and it's a simple fact that there are lots of C libraries based on `malloc` and `free` containing code that is very much worth reusing. When taking advantage of such a library, it's likely you'll end up with the responsibility for `free`ing memory `malloc`ed by the library and/or `malloc`ing memory the library itself will `free`. That's fine. There's nothing wrong with calling `malloc` and `free` inside a C++ program as long as you make sure the pointers you get from `malloc` always meet their maker in `free` and the pointers you get from `new` eventually find their way to `delete`. The problems start when you get sloppy and try to mix `new` with `free` or `malloc` with `delete`. That's just asking for trouble.

Given that `malloc` and `free` are ignorant of constructors and destructors and that mixing `malloc`/`free` with `new`/`delete` can be more volatile than a fraternity rush party, you're best off sticking to an exclusive diet of `new`s and `delete`s whenever you can.

Back to Item 2: Prefer <iostream> to <stdio.h>.
Continue to Item 4: Prefer C++-style comments.

Back to Item 3: Prefer new and delete to malloc and free.
Continue to Memory Management

Item 4: Prefer C++-style comments.

The good old C comment syntax works in C++ too, but the newfangled C++ comment-to-end-of-line syntax has some distinct advantages. For example, consider this situation:

```
if ( a > b ) {
  // int temp = a;    // swap a and b
  // a = b;
  // b = temp;
}
```

Here you have a code block that has been commented out for some reason or other, but in a stunning display of software engineering, the programmer who originally wrote the code actually included a comment to indicate what was going on. When the C++ comment form was used to comment out the block, the embedded comment was of no concern, but there could have been a serious problem had everybody chosen to use C-style comments:

```
if ( a > b ) {
  /*  int temp = a;  /* swap a and b */
      a = b;
      b = temp;
  */
}
```

Notice how the embedded comment inadvertently puts a premature end to the comment that is supposed to comment out the code block.

C-style comments still have their place. For example, they're invaluable in header files that are processed by both C and C++ compilers. Still, if you can use C++-style comments, you are often better off doing so.

It's worth pointing out that retrograde preprocessors that were written only for C don't know how to cope with C++-style comments, so things like the following sometimes don't work as expected:

```
#define LIGHT_SPEED   3e8    // m/sec (in a vacuum)
```

Given a preprocessor unfamiliar with C++, the comment at the end of the line becomes *part of the macro!* Of course, as is discussed in Item 1, you shouldn't be using the preprocessor to define

constants anyway.

Back to Item 3: Prefer new and delete to malloc and free.
Continue to Memory Management

Back to Item 4: Prefer C++-style comments.
Continue to Item 5: Use the same form in corresponding uses of new and delete.

Memory Management

Memory management concerns in C++ fall into two general camps: getting it right and making it perform efficiently. Good programmers understand that these concerns should be addressed in that order, because a program that is dazzlingly fast and astoundingly small is of little use if it doesn't behave the way it's supposed to. For most programmers, getting things right means calling memory allocation and deallocation routines correctly. Making things perform efficiently, on the other hand, often means writing custom versions of the allocation and deallocation routines. Getting things right there is even more important.

On the correctness front, C++ inherits from C one of its biggest headaches, that of potential memory leaks. Even virtual memory, wonderful invention though it is, is finite, and not everybody has virtual memory in the first place.

In C, a memory leak arises whenever memory allocated through `malloc` is never returned through `free`. The names of the players in C++ are `new` and `delete`, but the story is much the same. However, the situation is improved somewhat by the presence of destructors, because they provide a convenient repository for calls to `delete` that all objects must make when they are destroyed. At the same time, there is more to worry about, because `new` implicitly calls constructors and `delete` implicitly calls destructors. Furthermore, there is the complication that you can define your own versions of `operator new` and `operator delete`, both inside and outside of classes. This gives rise to all kinds of opportunities to make mistakes. The following Items (as well as Item M8) should help you avoid some of the most common ones.

Back to Item 4: Prefer C++-style comments.
Continue to Item 5: Use the same form in corresponding uses of new and delete.

Item 5: Use the same form in corresponding uses of `new` and `delete`.

What's wrong with this picture?

```
string *stringArray = new string[100];

...

delete stringArray;
```

Everything here appears to be in order — the use of `new` is matched with a use of `delete` — but something is still quite wrong: your program's behavior is undefined. At the very least, 99 of the 100 `string` objects pointed to by `stringArray` are unlikely to be properly destroyed, because their destructors will probably never be called.

When you use `new`, two things happen. First, memory is allocated (via the function `operator new`, about which I'll have more to say in Items 7-10 as well as Item M8). Second, one or more constructors are called for that memory. When you use `delete`, two other things happen: one or more destructors are called for the memory, then the memory is deallocated (via the function `operator delete` — see Items 8 and M8). The big question for `delete` is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called.

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? The only way for `delete` to know is for you to tell it. If you don't use brackets in your use of `delete`, `delete` assumes a single object is pointed to. Otherwise, it assumes that an array is pointed to:

```
string *stringPtr1 = new string;

string *stringPtr2 = new string[100];
```

```
...


delete stringPtr1;           // delete an object


delete [] stringPtr2;        // delete an array of
                             // objects
```

What would happen if you used the "`[]`" form on `stringPtr1`? The result is undefined. What would happen if you didn't use the "`[]`" form on `stringPtr2`? Well, that's undefined too. Furthermore, it's undefined even for built-in types like `int`s, even though such types lack destructors. The rule, then, is simple: if you use `[]` when you call `new`, you must use `[]` when you call `delete`. If you don't use `[]` when you call `new`, don't use `[]` when you call `delete`.

This is a particularly important rule to bear in mind when you are writing a class containing a pointer data member and also offering multiple constructors, because then you've got to be careful to use the *same form* of `new` in all the constructors to initialize the pointer member. If you don't, how will you know what form of `delete` to use in your destructor? For a further examination of this issue, see Item 11.

This rule is also important for the `typedef`-inclined, because it means that a `typedef`'s author must document which form of `delete` should be employed when `new` is used to conjure up objects of the `typedef` type. For example, consider this `typedef`:

```
typedef string AddressLines[4];      // a person's address
                                     // has 4 lines, each of
                                     // which is a string
```

Because `AddressLines` is an array, this use of `new`,

```
string *pal = new AddressLines;      // note that "new
                                     // AddressLines" returns
                                     // a string*, just like
                                     // "new string[4]" would
```

must be matched with the *array* form of `delete`:

```
delete pal;                         // undefined!


delete [] pal;                      // fine
```

To avoid such confusion, you're probably best off abstaining from `typedefs` for array types. That should be easy, however, because the standard C++ library (see Item 49) includes `string` and `vector` templates that reduce the need for built-in arrays to nearly zero. Here, for example, `AddressLines` could be defined to be a `vector` of `strings`. That is, `AddressLines` could be of type `vector<string>`.

Back to Memory Management
Continue to Item 6: Use delete on pointer members in destructors.

Back to Item 5: Use the same form in corresponding uses of new and delete.
Continue to Item 7: Be prepared for out-of-memory conditions.

Item 6: Use `delete` on pointer members in destructors.

Most of the time, classes performing dynamic memory allocation will use `new` in the constructor(s) to allocate the memory and will later use `delete` in the destructor to free up the memory. This isn't too difficult to get right when you first write the class, provided, of course, that you remember to employ `delete` on *all* the members that could have been assigned memory in *any* constructor.

However, the situation becomes more difficult as classes are maintained and enhanced, because the programmers making the modifications to the class may not be the ones who wrote the class in the first place. Under those conditions, it's easy to forget that adding a pointer member almost always requires each of the following:

- Initialization of the pointer in each of the constructors. If no memory is to be allocated to the pointer in a particular constructor, the pointer should be initialized to 0 (i.e., the null pointer).
- Deletion of the existing memory and assignment of new memory in the assignment operator. (See also Item 17.)
- Deletion of the pointer in the destructor.

If you forget to initialize a pointer in a constructor, or if you forget to handle it inside the assignment operator, the problem usually becomes apparent fairly quickly, so in practice those issues don't tend to plague you. Failing to delete the pointer in the destructor, however, often exhibits no obvious external symptoms. Instead, it manifests itself as a subtle memory leak, a slowly growing cancer that will eventually devour your address space and drive your program to an early demise. Because this particular problem doesn't usually call attention to itself, it's important that you keep it in mind whenever you add a pointer member to a class.

Note, by the way, that deleting a null pointer is always safe (it does nothing). Thus, if you write your constructors, your assignment operators, and your other member functions such that each pointer member of the class is always either pointing to valid memory or is null, you can merrily `delete` away in the destructor without regard for whether you ever used `new` for the pointer in question.

There's no reason to get fascist about this Item. For example, you certainly don't want to use `delete` on a pointer that wasn't initialized via `new`, and, except in the case of smart pointer objects (see Item M28), you almost *never* want to delete a pointer that was passed to you in the first place. In other words, your class destructor usually shouldn't be using `delete` unless your class members were the ones who used `new` in the first place.

Speaking of smart pointers, one way to avoid the need to delete pointer members is to replace those members with smart pointer objects like the standard C++ Library's `auto_ptr`. To see how this can work, take a look at Items M9 and M10.

Back to Item 5: Use the same form in corresponding uses of new and delete.

Continue to Item 7: Be prepared for out-of-memory conditions.

Back to Item 6: Use delete on pointer members in destructors.
Continue to Item 8: Adhere to convention when writing operator new and operator delete.

Item 7: Be prepared for out-of-memory conditions.

When `operator new` can't allocate the memory you request, it throws an exception. (It used to return 0, and some older compilers still do that. You can make your compilers do it again if you want to, but I'll defer that discussion until the end of this Item.) Deep in your heart of hearts, you know that handling out-of-memory exceptions is the only truly moral course of action. At the same time, you are keenly aware of the fact that doing so is a pain in the neck. As a result, chances are that you omit such handling from time to time. Like always, perhaps. Still, you must harbor a lurking sense of guilt. I mean, what if `new` really *does* yield an exception?

You may think that one reasonable way to cope with this matter is to fall back on your days in the gutter, i.e., to use the preprocessor. For example, a common C idiom is to define a type-independent macro to allocate memory and then check to make sure the allocation succeeded. For C++, such a macro might look something like this:

```
#define NEW(PTR, TYPE)                          \
    try { (PTR) = new TYPE; }                   \
    catch (std::bad_alloc&) { assert(0); }
```

("Wait! What's this `std::bad_alloc` business?", you ask. `bad_alloc` is the type of exception `operator new` throws when it can't satisfy a memory allocation request, and `std` is the name of the namespace (see Item 28) where `bad_alloc` is defined. "Okay," you continue, "what's this `assert` business?" Well, if you look in the standard C include file `<assert.h>` (or its namespace-savvy C++ equivalent, `<cassert>` — see Item 49), you'll find that `assert` is a macro. The macro checks to see if the expression it's passed is non-zero, and, if it's not, it issues an error message and calls `abort`. Okay, it does that only when the standard macro `NDEBUG` isn't defined, i.e., in debug mode. In production mode, i.e., when `NDEBUG` is defined, `assert` expands to nothing — to a `void` statement. You thus check `assertions` only when debugging.)

This `NEW` macro suffers from the common error of using an `assert` to test a condition that might occur in production code (after all, you can run out of memory at any time), but it also has a drawback specific to C++: it fails to take into account the myriad ways in which `new` can be used. There are three common syntactic forms for getting new objects of type `T`, and you need to deal with the possibility of exceptions for each of these forms:

```
new T;
```

```
new T(constructor arguments);


new T[size];
```

This oversimplifies the problem, however, because clients can define their own (overloaded) versions of `operator new`, so programs may contain an arbitrary number of different syntactic forms for using `new`.

How, then, to cope? If you're willing to settle for a very simple error-handling strategy, you can set things up so that if a request for memory cannot be satisfied, an error-handling function you specify is called. This strategy relies on the convention that when `operator new` cannot satisfy a request, it calls a client-specifiable error-handling function — often called a *new-handler* — before it throws an exception. (In truth, what `operator new` really does is slightly more complicated. Details are provided in Item 8.)

To specify the out-of-memory-handling function, clients call `set_new_handler`, which is specified in the header `<new>` more or less like this:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

As you can see, `new_handler` is a typedef for a pointer to a function that takes and returns nothing, and `set_new_handler` is a function that takes and returns a `new_handler`.

`set_new_handler`'s parameter is a pointer to the function `operator new` should call if it can't allocate the requested memory. The return value of `set_new_handler` is a pointer to the function in effect for that purpose before `set_new_handler` was called.

You use `set_new_handler` like this:

```
// function to call if operator new can't allocate enough memory
void noMoreMemory()
{
  cerr << "Unable to satisfy request for memory\n";
  abort();
}


int main()
{
  set_new_handler(noMoreMemory);
```

```
  int *pBigDataArray = new int[100000000];

  ...

}
```

If, as seems likely, `operator new` is unable to allocate space for 100,000,000 integers, `noMoreMemory` will be called, and the program will abort after issuing an error message. This is a marginally better way to terminate the program than a simple core dump. (By the way, consider what happens if memory must be dynamically allocated during the course of writing the error message to `cerr`...)

When `operator new` cannot satisfy a request for memory, it calls the new-handler function not once, but *repeatedly* until it can find enough memory. The code giving rise to these repeated calls is shown in Item 8, but this high-level description is enough to conclude that a well-designed new-handler function must do one of the following:

- **Make more memory available.** This may allow `operator new`'s next attempt to allocate the memory to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it the first time the new-handler is invoked. Such a release is often accompanied by some kind of warning to the user that memory is low and that future requests may fail unless more memory is somehow made available.
- **Install a different new-handler.** If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that is more resourceful. If so, the current new-handler can install the other new-handler in its place (by calling `set_new_handler`). The next time `operator new` calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static or global data that affects the new-handler's behavior.)
- **Deinstall the new-handler,** i.e., pass the null pointer to `set_new_handler`. With no new-handler installed, `operator new` will throw an exception of type `std::bad_alloc` when its attempt to allocate memory is unsuccessful.
- **Throw an exception** of type `std::bad_alloc` or some type derived from `std::bad_alloc`. Such exceptions will not be caught by `operator new`, so they will propagate to the site originating the request for memory. (Throwing an exception of a different type will violate `operator new`'s exception specification. The default action when that happens is to call `abort`, so if your new-handler is going to throw an exception, you definitely want to make sure it's from the `std::bad_alloc` hierarchy. For more information on exception specifications, see Item M14.)
- **Not return**, typically by calling `abort` or `exit`, both of which are found in the standard C

library (and thus in the standard C++ library — see Item 49).

These choices give you considerable flexibility in implementing new-handler functions.

Sometimes you'd like to handle memory allocation failures in different ways, depending on the class of the object being allocated:

```
class X {
public:
  static void outOfMemory();


  ...


};


class Y {
public:
  static void outOfMemory();


  ...


};



X* p1 = new X;      // if allocation is unsuccessful,
                    // call X::outOfMemory



Y* p2 = new Y;      // if allocation is unsuccessful,
                    // call Y::outOfMemory
```

C++ has no support for class-specific new-handlers, but it doesn't need to. You can implement this behavior yourself. You just have each class provide its own versions of `set_new_handler` and `operator new`. The class's `set_new_handler` allows clients to specify the new-handler for the class (just like the standard `set_new_handler` allows clients to specify the global new-handler). The class's `operator new` ensures that the class-specific new-handler is used in place of the global new-handler when memory for class objects is allocated.

Consider a class X for which you want to handle memory allocation failures. You'll have to keep track of the function to call when operator new can't allocate enough memory for an object of type X, so you'll declare a static member of type new_handler to point to the new-handler function for the class. Your class X will look something like this:

```
class X {
public:
  static new_handler set_new_handler(new_handler p);
  static void * operator new(size_t size);


private:
  static new_handler currentHandler;
};
```

Static class members must be defined outside the class definition. Because you'll want to use the default initialization of static objects to 0, you'll define X::currentHandler without initializing it:

```
new_handler X::currentHandler;      // sets currentHandler
                                    // to 0 (i.e., null) by
                                    // default
```

The set_new_handler function in class X will save whatever pointer is passed to it. It will return whatever pointer had been saved prior to the call. This is exactly what the standard version of set_new_handler does:

```
new_handler X::set_new_handler(new_handler p)
{
  new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

Finally, X's operator new will do the following:

1. Call the standard set_new_handler with X's error-handling function. This will install X's new-handler as the global new- handler. In the code below, notice how you explicitly reference the std scope (where the standard set_new_handler resides) by using the "::" notation.
2. Call the global operator new to actually allocate the requested memory. If the initial attempt at allocation fails, the global operator new will invoke X's new-handler, because that function was just installed as the global new-handler. If the global operator new is ultimately unable to

find a way to allocate the requested memory, it will throw a `std::bad_alloc` exception, which `X`'s `operator new` will catch. `X`'s `operator new` will then restore the global new-handler that was originally in place, and it will return by propagating the exception.

3. Assuming the global `operator new` was able to successfully allocate enough memory for an object of type `X`, `X`'s `operator new` will again call the standard `set_new_handler` to restore the global error-handling function to what it was originally. It will then return a pointer to the allocated memory.

Here's how you say all that in C++:

```
void * X::operator new(size_t size)
{
  new_handler globalHandler =                // install X's
    std::set_new_handler(currentHandler);    // handler



  void *memory;



  try {                                      // attempt
    memory = ::operator new(size);           // allocation
  }
  catch (std::bad_alloc&) {                  // restore
    std::set_new_handler(globalHandler);     // handler;
    throw;                                   // propagate
  }                                          // exception



  std::set_new_handler(globalHandler);       // restore
                                             // handler
  return memory;
}
```

If the duplicated calls to `std::set_new_handler` caught your eye, turn to Item M9 for information on how to eliminate them.

Clients of class `X` use its new-handling capabilities like this:

```
void noMoreMemory();                           // decl. of function to
                                               // call if memory allocation
                                               // for X objects fails
```

```
X::set_new_handler(noMoreMemory);
                                        // set noMoreMemory as X's
                                        // new-handling function

X *px1 = new X;                         // if memory allocation
                                        // fails, call noMoreMemory

string *ps = new string;                // if memory allocation
                                        // fails, call the global
                                        // new-handling function
                                        // (if there is one)

X::set_new_handler(0);                  // set the X-specific
                                        // new-handling function
                                        // to nothing (i.e., null)

X *px2 = new X;                         // if memory allocation
                                        // fails, throw an exception
                                        // immediately. (There is
                                        // no new-handling function
                                        // for class X.)
```

You may note that the code for implementing this scheme is the same regardless of the class, so a reasonable inclination would be to reuse it in other places. As Item 41 explains, both inheritance and templates can be used to create reusable code. However, in this case, it's a combination of the two that gives you what you need.

All you have to do is create a "mixin-style" base class, i.e., a base class that's designed to allow derived classes to inherit a single specific capability — in this case, the ability to set a class-specific new-handler. Then you turn the base class into a template. The base class part of the design lets derived classes inherit the `set_new_handler` and `operator new` functions they all need, while the template part of the design ensures that each inheriting class gets a different `currentHandler` data member. The result may sound a little complicated, but you'll find that the code looks reassuringly familiar. In fact, about the only real difference is that it's now reusable by any class that wants it:

```
template<class T>    // "mixin-style" base class
class NewHandlerSupport {   // for class-specific
public:      // set_new_handler support



  static new_handler set_new_handler(new_handler p);
  static void * operator new(size_t size);



private:
  static new_handler currentHandler;
};



template<class T>
new_handler NewHandlerSupport<T>::set_new_handler(new_handler p)
{
  new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}



template<class T>
void * NewHandlerSupport<T>::operator new(size_t size)
{
  new_handler globalHandler =
    std::set_new_handler(currentHandler);



  void *memory;



  try {
    memory = ::operator new(size);
  }
  catch (std::bad_alloc&) {
    std::set_new_handler(globalHandler);
    throw;
  }



  std::set_new_handler(globalHandler);



  return memory;
}
```

```
// this sets each currentHandler to 0
template<class T>
new_handler NewHandlerSupport<T>::currentHandler;
```

With this class template, adding `set_new_handler` support to class `X` is easy: `X` just inherits from `newHandlerSupport<X>`:

```
// note inheritance from mixin base class template. (See
// my article on counting objects for information on why
// private inheritance might be preferable here.)
class X: public NewHandlerSupport<X> {


  ...                 // as before, but no declarations for
};                    // set_new_handler or operator new
```

Clients of `X` remain oblivious to all the behind-the-scenes action; their old code continues to work. This is good, because one thing you can usually rely on your clients being is oblivious.

Using `set_new_handler` is a convenient, easy way to cope with the possibility of out-of-memory conditions. Certainly it's a lot more attractive than wrapping every use of `new` inside a `try` block. Furthermore, templates like `NewHandlerSupport` make it simple to add a class-specific new-handler to any class that wants one. Mixin-style inheritance, however, invariably leads to the topic of multiple inheritance, and before starting down that slippery slope, you'll definitely want to read Item 43.

Until 1993, C++ required that `operator new` return 0 when it was unable to satisfy a memory request. The current behavior is for `operator new` to throw a `std::bad_alloc` exception, but a lot of C++ was written before compilers began supporting the revised specification. The ◦C++ standardization committee didn't want to abandon the established test-for-0 code base, so they provided alternative forms of `operator new` (and `operator new[]` — see Item 8) that continue to offer the traditional failure-yields-0 behavior. These forms are called "nothrow" forms because, well, they never do a `throw`, and they employ `nothrow` objects (defined in the standard header `<new>`) at the point where `new` is used:

```
class Widget { ... };


Widget *pw1 = new Widget;       // throws std::bad_alloc if
                                // allocation fails
```

```
if (pw1 == 0) ...              // this test must fail



Widget *pw2 =
  new (nothrow) Widget;        // returns 0 if allocation
                               // fails



if (pw2 == 0) ...              // this test may succeed
```

Regardless of whether you use "normal" (i.e., exception-throwing) `new` or "nothrow" `new`, it's important that you be prepared to handle memory allocation failures. The easiest way to do that is to take advantage of `set_new_handler`, because it works with both forms.

Back to Item 6: Use delete on pointer members in destructors.
Continue to Item 8: Adhere to convention when writing operator new and operator delete.

Back to Item 7: Be prepared for out-of-memory conditions.
Continue to Item 9: Avoid hiding the "normal" form of new.

Item 8: Adhere to convention when writing `operator new` and `operator delete`.

When you take it upon yourself to write `operator new` (Item 10 explains why you might want to), it's important that your function(s) offer behavior that is consistent with the default `operator new`. In practical terms, this means having the right return value, calling an error-handling function when insufficient memory is available (see Item 7), and being prepared to cope with requests for no memory. You also need to avoid inadvertently hiding the "normal" form of `new`, but that's a topic for Item 9.

The return value part is easy. If you can supply the requested memory, you just return a pointer to it. If you can't, you follow the rule described in Item 7 and throw an exception of type `std::bad_alloc`.

It's not quite that simple, however, because `operator new` actually tries to allocate memory more than once, calling the error-handling function after each failure, the assumption being that the error-handling function might be able to do something to free up some memory. Only when the pointer to the error-handling function is null does `operator new` throw an exception.

In addition, the ◦C++ standard requires that `operator new` return a legitimate pointer even when 0 bytes are requested. (Believe it or not, requiring this odd-sounding behavior actually simplifies things elsewhere in the language.)

That being the case, pseudocode for a non-member `operator new` looks like this:

```
void * operator new(size_t size)        // your operator new might
{                                       // take additional params



  if (size == 0) {                      // handle 0-byte requests
    size = 1;                           // by treating them as
  }                                     // 1-byte requests


  while (1) {
    attempt to allocate size bytes;
```

```
      if (the allocation was successful)
        return (a pointer to the memory);



      // allocation was unsuccessful; find out what the
      // current error-handling function is (see Item 7)
      new_handler globalHandler = set_new_handler(0);
      set_new_handler(globalHandler);



      if (globalHandler) (*globalHandler)();
      else throw std::bad_alloc();
    }
  }
```

The trick of treating requests for zero bytes as if they were really requests for one byte looks slimy, but it's simple, it's legal, it works, and how often do you expect to be asked for zero bytes, anyway?

You may also look askance at the place in the pseudocode where the error-handling function pointer is set to null, then promptly reset to what it was originally. Unfortunately, there is no way to get at the error-handling function pointer directly, so you have to call `set_new_handler` to find out what it is. Crude, yes, but also effective.

Item 7 remarks that `operator new` contains an infinite loop, and the code above shows that loop explicitly — `while (1)` is about as infinite as it gets. The only way out of the loop is for memory to be successfully allocated or for the new-handling function to do one of the things described in Item 7: make more memory available, install a different new-handler, deinstall the new-handler, throw an exception of or derived from `std::bad_alloc`, or fail to return. It should now be clear why the new-handler must do one of those things. If it doesn't, the loop inside `operator new` will never terminate.

One of the things many people don't realize about `operator new` is that it's inherited by subclasses. That can lead to some interesting complications. In the pseudocode for `operator new` above, notice that the function tries to allocate `size` bytes (unless `size` is 0). That makes perfect sense, because that's the argument that was passed to the function. However, most class-specific versions of `operator new` (including the one you'll find in Item 10) are designed for a *specific* class, *not* for a class *or* any of its subclasses. That is, given an `operator new` for a class `X`, the behavior of that function is almost always carefully tuned for objects of size `sizeof(X)` — nothing larger and nothing smaller. Because of inheritance, however, it is possible that the `operator new` in a base class will be called to allocate memory for an object of a derived class:

```
class Base {
public:
```

```
  static void * operator new(size_t size);
  ...
};



class Derived: public Base       // Derived doesn't declare
{ ... };                         // operator new



Derived *p = new Derived;        // calls Base::operator new!
```

If `Base`'s class-specific `operator new` wasn't designed to cope with this — and chances are slim that it was — the best way for it to handle the situation is to slough off calls requesting the "wrong" amount of memory to the standard `operator new`, like this:

```
void * Base::operator new(size_t size)
{
  if (size != sizeof(Base))              // if size is "wrong,"
    return ::operator new(size);         // have standard operator
                                         // new handle the request



  ...                                    // otherwise handle
                                         // the request here
}
```

"Hold on!" I hear you cry, "You forgot to check for the pathological-but-nevertheless-possible case where `size` is zero!" Actually, I didn't, and please stop using hyphens when you cry out. The test is still there, it's just been incorporated into the test of `size` against `sizeof(Base)`. The ◦C++ standard works in mysterious ways, and one of those ways is to decree that all freestanding classes have nonzero size. By definition, `sizeof(Base)` can never be zero (even if it has no members), so if `size` is zero, the request will be forwarded to `::operator new`, and it will become that function's responsibility to treat the request in a reasonable fashion. (Interestingly, `sizeof(Base)` may be zero if `Base` is not a freestanding class. For details, consult my article on counting objects.)

If you'd like to control memory allocation for arrays on a per-class basis, you need to implement `operator new`'s array-specific cousin, `operator new[]`. (This function is usually called "array new," because it's hard to figure out how to pronounce "operator new[]".) If you decide to write `operator new[]`, remember that all you're doing is allocating raw memory — you can't do anything to the as-yet-nonexistent objects in the array. In fact, you can't even figure out how many objects will be in the array, because you don't know how big each object is. After all, a base class's `operator new[]`

might, through inheritance, be called to allocate memory for an array of derived class objects, and derived class objects are usually bigger than base class objects. Hence, you can't assume inside `Base::operator new[]` that the size of each object going into the array is `sizeof(Base)`, and that means you can't assume that the number of objects in the array is `(`*`bytes requested`*`)/sizeof(Base)`. For more information on `operator new[]`, see Item M8.

So much for the conventions you need to follow when writing `operator new` (and `operator new[]`). For `operator delete` (and its array counterpart, `operator delete[]`), things are simpler. About all you need to remember is that C++ guarantees it's always safe to delete the null pointer, so you need to honor that guarantee. Here's pseudocode for a non-member `operator delete`:

```
void operator delete(void *rawMemory)
{
  if (rawMemory == 0) return;    // do nothing if the null
                                 // pointer is being deleted


  deallocate the memory pointed to by rawMemory;


  return;
}
```

The member version of this function is simple, too, except you've got to be sure to check the size of what's being deleted. Assuming your class-specific `operator new` forwards requests of the "wrong" size to `::operator new`, you've got to forward "wrongly sized" deletion requests to `::operator delete`:

```
class Base {                        // same as before, but now
public:                             // op. delete is declared
  static void * operator new(size_t size);
  static void operator delete(void *rawMemory, size_t size);
  ...
};


void Base::operator delete(void *rawMemory, size_t size)
{
  if (rawMemory == 0) return;       // check for null pointer


  if (size != sizeof(Base)) {       // if size is "wrong,"
    ::operator delete(rawMemory);   // have standard operator
    return;                         // delete handle the request
```

```
  }

  deallocate the memory pointed to by rawMemory;

  return;
}
```

The conventions, then, for `operator new` and `operator delete` (and their array counterparts) are not particularly onerous, but it is important that you obey them. If your allocation routines support new-handler functions and correctly deal with zero-sized requests, you're all but finished, and if your deallocation routines cope with null pointers, there's little more to do. Add support for inheritance in member versions of the functions, and *presto!* — you're done.

Back to Item 7: Be prepared for out-of-memory conditions.
Continue to Item 9: Avoid hiding the "normal" form of new.

Back to Item 8: Adhere to convention when writing operator new and operator delete.
Continue to Item 10: Write operator delete if you write operator new.

Item 9: Avoid hiding the "normal" form of **new**.

A declaration of a name in an inner scope hides the same name in outer scopes, so for a function `f` at both global and class scope, the member function will hide the global function:

```
void f();                               // global function


class X {
public:
  void f();                             // member function
};


X x;



f();                                    // calls global f



x.f();                                  // calls X::f
```

This is unsurprising and normally causes no confusion, because global and member functions are usually invoked using different syntactic forms. However, if you add to this class an `operator new` taking additional parameters, the result is likely to be an eye-opener:

```
class X {
public:
  void f();



  // operator new allowing specification of a
  // new-handling function
  static void * operator new(size_t size, new_handler p);
};



void specialErrorHandler();          // definition is elsewhere
```

```
X *px1 =
  new (specialErrorHandler) X;       // calls X::operator new



X *px2 = new X;                      // error!
```

By declaring a function called "operator new" inside the class, you inadvertently block access to the "normal" form of `new`. Why this is so is discussed in Item 50. Here we're more interested in figuring out how to avoid the problem.

One solution is to write a class-specific `operator new` that supports the "normal" invocation form. If it does the same thing as the global version, that can be efficiently and elegantly encapsulated as an inline function:

```
class X {
public:
  void f();



  static void * operator new(size_t size, new_handler p);



  static void * operator new(size_t size)
  { return ::operator new(size); }
};



X *px1 =
  new (specialErrorHandler) X;      // calls X::operator
                                    // new(size_t, new_handler)



X* px2 = new X;                     // calls X::operator
                                    // new(size_t)
```

An alternative is to provide a default parameter value (see Item 24) for each additional parameter you add to `operator new`:

```
class X {
public:
```

```
  void f();


  static
    void * operator new(size_t size,                // note default
                        new_handler p = 0);         // value for p
};



X *px1 = new (specialErrorHandler) X;               // fine


X* px2 = new X;                                     // also fine
```

Either way, if you later decide to customize the behavior of the "normal" form of `new`, all you need to do is rewrite the function; callers will get the customized behavior automatically when they relink.

Back to Item 8: Adhere to convention when writing operator new and operator delete.
Continue to Item 10: Write operator delete if you write operator new.

Back to Item 9: Avoid hiding the "normal" form of new.
Continue to Constructors, Destructors, and Assignment Operators

Item 10: Write `operator delete` if you write `operator new`.

Let's step back for a moment and return to fundamentals. Why would anybody want to write their own version of `operator new` or `operator delete` in the first place?

More often than not, the answer is efficiency. The default versions of `operator new` and `operator delete` are perfectly adequate for general-purpose use, but their flexibility inevitably leaves room for improvements in their performance in a more circumscribed context. This is especially true for applications that dynamically allocate a large number of small objects.

As an example, consider a class for representing airplanes, where the `Airplane` class contains only a pointer to the actual representation for airplane objects (a technique discussed in Item 34):

```
class AirplaneRep { ... };      // representation for an
                                // Airplane object
class Airplane {
public:
  ...
private:
  AirplaneRep *rep;             // pointer to representation
};
```

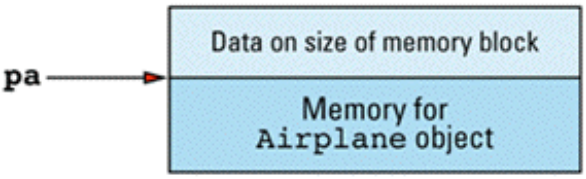
An `Airplane` object is not very big; it contains but a single pointer. (As explained in Items 14 and M24, it may implicitly contain a second pointer if the `Airplane` class declares virtual functions.) When you allocate an `Airplane` object by calling `operator new`, however, you probably get back more memory than is needed to store this pointer (or pair of pointers). The reason for this seemingly wayward behavior has to do with the need for `operator new` and `operator delete` to communicate with one another.

Because the default version of `operator new` is a general-purpose allocator, it must be prepared to allocate blocks of any size. Similarly, the default version of `operator delete` must be prepared to deallocate blocks of whatever size `operator new` allocated. For `operator delete` to know how much memory to deallocate, it must have some way of knowing how much memory `operator new` allocated in the first place. A common way for `operator new` to tell `operator delete` how much memory it allocated is by prepending to the memory it returns some additional data that specifies the size of the allocated block. That is, when you say this,

```
Airplane *pa = new Airplane;
```

you don't necessarily get back a block of memory that looks like this:

Instead, you often get back a block of memory that looks more like this:



For small objects like those of class `Airplane`, this additional bookkeeping data can more than double the amount of memory needed for each dynamically allocated object (especially if the class contains no virtual functions).

If you're developing software for an environment in which memory is precious, you may not be able to afford this kind of spendthrift allocation. By writing your own `operator new` for the `Airplane` class, you can take advantage of the fact that all `Airplane` objects are the same size, so there isn't any need for bookkeeping information to be kept with each allocated block.

One way to implement your class-specific `operator new` is to ask the default `operator new` for big blocks of raw memory, each block of sufficient size to hold a large number of `Airplane` objects. The memory chunks for `Airplane` objects themselves will be taken from these big blocks. Currently unused chunks will be organized into a linked list — the *free* list — of chunks that are available for future `Airplane` use. This may make it sound like you'll have to pay for the overhead of a `next` field in every object (to support the list), but you won't: the space for the `rep` field (which is necessary only for memory chunks in use as `Airplane` objects) will also serve as the place to store the `next` pointer (because that pointer is needed only for chunks of memory *not* in use as `Airplane` objects). You'll arrange for this job-sharing in the usual fashion: you'll use a `union`.

To turn this design into reality, you have to modify the definition of `Airplane` to support custom memory management. You do it as follows:

```
class Airplane {          // modified class — now supports
public:                   // custom memory management
```

```
  static void * operator new(size_t size);


  ...


private:
  union {
    AirplaneRep *rep;      // for objects in use
    Airplane *next;        // for objects on free list
  };



  // this class-specific constant (see Item 1) specifies how
  // many Airplane objects fit into a big memory block;
  // it's initialized below
  static const int BLOCK_SIZE;


  static Airplane *headOfFreeList;


};
```

Here you've added the declarations for `operator new`, the union that allows the `rep` and `next` fields to occupy the same memory, a class-specific constant for specifying how big each allocated block should be, and a static pointer to keep track of the head of the free list. It's important to use a static member for this last task, because there's one free list for the entire *class*, not one free list for each `Airplane` *object*.

The next thing to do is to write the new `operator new`:

```
void * Airplane::operator new(size_t size)
{
  // send requests of the "wrong" size to ::operator new();
  // for details, see Item 8
  if (size != sizeof(Airplane))
    return ::operator new(size);



  Airplane *p =           // p is now a pointer to the
    headOfFreeList;       // head of the free list
```

```
  // if p is valid, just move the list head to the
  // next element in the free list
  if (p)
    headOfFreeList = p->next;



  else {
    // The free list is empty. Allocate a block of memory
    // big enough to hold BLOCK_SIZE Airplane objects
    Airplane *newBlock =
      static_cast<Airplane*>(::operator new(BLOCK_SIZE *
                                            sizeof(Airplane)));



    // form a new free list by linking the memory chunks
    // together; skip the zeroth element, because you'll
    // return that to the caller of operator new
    for (int i = 1; i < BLOCK_SIZE-1; ++i)
      newBlock[i].next = &newBlock[i+1];



    // terminate the linked list with a null pointer
    newBlock[BLOCK_SIZE-1].next = 0;



    // set p to front of list, headOfFreeList to
    // chunk immediately following
    p = newBlock;
    headOfFreeList = &newBlock[1];
  }



  return p;
}
```

If you've read Item 8, you know that when `operator new` can't satisfy a request for memory, it's supposed to perform a series of ritualistic steps involving new-handler functions and exceptions. There is no sign of such steps above. That's because this `operator new` gets all the memory it manages from `::operator new`. That means this `operator new` can fail only if `::operator new` does. But if `::operator new` fails, *it* must engage in the new-handling ritual (possibly culminating in the throwing of an exception), so there is no need for `Airplane`'s `operator new` to do it, too. In other words, the new-handler behavior is there, you just don't see it, because it's hidden inside `::operator new`.

Given this `operator new`, the only thing left to do is provide the obligatory definitions of `Airplane`'s static data members:

```
Airplane *Airplane::headOfFreeList;         // these definitions
                                            // go in an implemen-
const int Airplane::BLOCK_SIZE = 512;       // tation file, not
                                            // a header file
```

There's no need to explicitly set `headOfFreeList` to the null pointer, because static members are initialized to 0 by default. The value for `BLOCK_SIZE`, of course, determines the size of each memory block we get from `::operator new`.

This version of `operator new` will work just fine. Not only will it use a lot less memory for `Airplane` objects than the default `operator new`, it's also likely to be faster, possibly as much as two orders of magnitude faster. That shouldn't be surprising. After all, the general version of `operator new` has to cope with memory requests of different sizes, has to worry about internal and external fragmentation, etc., whereas your version of `operator new` just manipulates a couple of pointers in a linked list. It's easy to be fast when you don't have to be flexible.

At long last we are in a position to discuss `operator delete`. Remember `operator delete`? This Item is *about* `operator delete`. As currently written, your `Airplane` class declares `operator new`, but it does not declare `operator delete`. Now consider what happens when a client writes the following, which is nothing if not eminently reasonable:

```
Airplane *pa = new Airplane;        // calls
                                    // Airplane::operator new
...



delete pa;                          // calls ::operator delete
```

If you listen closely when you read this code, you can hear the sound of an airplane crashing and burning, with much weeping and wailing by the programmers who knew it. The problem is that `operator new` (the one defined in `Airplane`) returns a pointer to memory *without any header information*, but `operator delete` (the default, global one) assumes that the memory it's passed *does* contain header information! Surely this is a recipe for disaster.

This example illustrates the general rule: `operator new` and `operator delete` must be written in concert so that they share the same assumptions. If you're going to roll your own memory allocation routine, be sure to roll one for deallocation, too. (For another reason why you should follow this advice, turn to the sidebar on placement `new` and placement `delete` in my article on counting

objects in C++.)

Here's how you solve the problem with the `Airplane` class:

```
class Airplane {        // same as before, except there's
public:                 // now a decl. for operator delete
  ...



  static void operator delete(void *deadObject,
                              size_t size);


};


// operator delete is passed a memory chunk, which,
// if it's the right size, is just added to the
// front of the list of free chunks
void Airplane::operator delete(void *deadObject,
                               size_t size)
{
  if (deadObject == 0) return;         // see Item 8



  if (size != sizeof(Airplane))     {  // see Item 8
    ::operator delete(deadObject);
    return;
  }


  Airplane *carcass =
    static_cast<Airplane*>(deadObject);


  carcass->next = headOfFreeList;
  headOfFreeList = carcass;
}
```

Because you were careful in `operator new` to ensure that calls of the "wrong" size were forwarded to the global `operator new` (see Item 8), you must demonstrate equal care in ensuring that such "improperly sized" objects are handled by the global version of `operator delete`. If you did not, you'd run into precisely the problem you have been laboring so arduously to avoid — a semantic

mismatch between `new` and `delete`.

Interestingly, the `size_t` value C++ passes to `operator delete` may be *incorrect* if the object being deleted was derived from a base class lacking a virtual destructor. This is reason enough for making sure your base classes have virtual destructors, but Item 14 describes a second, arguably better reason. For now, simply note that if you omit virtual destructors in base classes, `operator delete` functions may not work correctly.

All of which is well and good, but I can tell by the furrow in your brow that what you're really concerned about is the memory leak. With all the software development experience you bring to the table, there's no way you'd fail to notice that `Airplane`'s `operator new` calls `::operator new` to get big blocks of memory, but Airplane's `operator delete` fails to release those blocks.[4] *Memory leak! Memory leak!* I can almost hear the alarm bells going off in your head.

Listen to me carefully: *there is no memory leak*.

A memory leak arises when memory is allocated, then all pointers to that memory are lost. Absent garbage collection or some other extralinguistic mechanism, such memory cannot be reclaimed. But this design has no memory leak, because it's never the case that all pointers to memory are lost. Each big block of memory is first broken down into `Airplane`-sized chunks, and these chunks are then placed on the free list. When clients call `Airplane::operator new`, chunks are removed from the free list, and clients receive pointers to them. When clients call `operator delete`, the chunks are put back on the free list. With this design, all memory chunks are either in use as `Airplane` objects (in which case it's the clients' responsibility to avoid leaking their memory) or are on the free list (in which case there's a pointer to the memory). There is no memory leak.

Nevertheless, the blocks of memory returned by `::operator new` are never released by `Airplane::operator delete`, and there has to be *some* name for that. There is. You've created a memory *pool*. Call it semantic gymnastics if you must, but there is an important difference between a memory leak and a memory pool. A memory leak may grow indefinitely, even if clients are well-behaved, but a memory pool never grows larger than the maximum amount of memory requested by its clients.

It would not be difficult to modify `Airplane`'s memory management routines so that the blocks of memory returned by `::operator new` were automatically released when they were no longer in use, but there are two reasons why you might not want to do it.

The first concerns your likely motivation for tackling custom memory management. There are many reasons why you might do it, but the most common one is that you've determined (see Item M16) that the default `operator new` and `operator delete` use too much memory or are too slow (or both). That being the case, every additional byte and every additional statement you devote to tracking and releasing those big memory blocks comes straight off the bottom line: your software

runs slower and uses more memory than it would if you adopted the pool strategy. For libraries and applications in which performance is at a premium and you can expect pool sizes to be reasonably bounded, the pool approach may well be best.

The second reason has to do with pathological behavior. Suppose `Airplane`'s memory management routines are modified so `Airplane`'s `operator delete` releases any big block of memory that has no active objects in it. Now consider this program:

```
int main()
{
  Airplane *pa = new Airplane;     // first allocation: get big
                                   // block, make free list, etc.



  delete pa;                       // block is now empty;
                                   // release it



  pa = new Airplane;               // uh oh, get block again,
                                   // make free list, etc.



  delete pa;                       // okay, block is empty
                                   // again; release it



  ...                              // you get the idea...



  return 0;
}
```

This nasty little program will run slower and use more memory than with even the *default* `operator new` and `operator delete`, much less the pool-based versions of those functions!

Of course, there are ways to deal with this pathology, but the more you code for uncommon special cases, the closer you get to reimplementing the default memory management functions, and then what have you gained? A memory pool is not the answer to all memory management questions, but it's a reasonable answer to many of them.

In fact, it's a reasonable answer often enough that you may be bothered by the need to reimplement it for different classes. "Surely," you think to yourself, "there should be a way to package the notion of a fixed-sized memory allocator so it's easily reused." There is, though this Item has droned on long enough that I'll leave the details in the form of the dreaded exercise for the reader.

Instead, I'll simply show a minimal interface (see Item 18) to a *Pool* class, where each object of type `Pool` is an allocator for objects of the size specified in the `Pool`'s constructor:

```
class Pool {
public:
  Pool(size_t n);                       // Create an allocator for
                                        // objects of size n

  void * alloc(size_t n)  ;             // Allocate enough memory
                                        // for one object; follow
                                        // operator new conventions
                                        // from Item 8

  void free(  void *p, size_t n);       // Return to the pool the
                                        // memory pointed to by p;
                                        // follow operator delete
                                        // conventions from Item 8

  ~Pool();                              // Deallocate all memory in
                                        // the pool
};
```

This class allows `Pool` objects to be created, to perform allocation and deallocation operations, and to be destroyed. When a `Pool` object is destroyed, it releases all the memory it allocated. This means there is now a way to avoid the memory leak-like behavior that `Airplane`'s functions exhibited. However, this also means that if a `Pool`'s destructor is called too soon (before all the objects using its memory have been destroyed), some objects will find their memory yanked out from under them before they're done using it. To say that the resulting behavior is undefined is being generous.

Given this `Pool` class, even a Java programmer can add custom memory management capabilities to `Airplane` without breaking a sweat:

```
class Airplane {
```

```
public:

  ...                                 // usual Airplane functions

  static void * operator new(size_t size);
  static void operator delete(void *p, size_t size);

private:
  AirplaneRep *rep;                   // pointer to representation
  static Pool memPool;                // memory pool for Airplanes

};

inline void * Airplane::operator new(size_t size)
{ return memPool.alloc(size); }

inline void Airplane::operator delete(void *p,
                                      size_t size)
{ memPool.free(p, size); }

// create a new pool for Airplane objects; this goes in
// the class implementation file
Pool Airplane::memPool(sizeof(Airplane));
```

This is a much cleaner design than the one we saw earlier, because the `Airplane` class is no longer cluttered with non-airplane details. Gone are the `union`, the head of the free list, the constant defining how big each raw memory block should be, etc. That's all hidden inside `Pool`, which is really where it should be. Let `Pool`'s author worry about memory management minutiae. Your job is to make the `Airplane` class work properly.

Now, it's interesting to see how custom memory management routines can improve program performance, and it's worthwhile to see how such routines can be encapsulated inside a class like `Pool`, but let us not lose sight of the main point. That point is that `operator new` and `operator delete` need to work together, so if you write `operator new`, be sure to write `operator delete`, as well.

---

[4] I write this with certainty, because I failed to address this issue in the first edition of this book,

and *many* readers upbraided me for the omission. There's nothing quite like a few thousand proofreaders to demonstrate one's fallibility, sigh.
Return

Back to Item 9: Avoid hiding the "normal" form of new.
Continue to Constructors, Destructors, and Assignment Operators

Back to Item 10: Write operator delete if you write operator new.
Continue to Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.

Constructors, Destructors, and Assignment Operators

Almost every class you write will have one or more constructors, a destructor, and an assignment operator. Little wonder. These are your bread-and-butter functions, the ones that control the fundamental operations of bringing a new object into existence and making sure it's initialized; getting rid of an object and making sure it's been properly cleaned up; and giving an object a new value. Making mistakes in these functions will lead to far-reaching and distinctly unpleasant repercussions throughout your classes, so it's vital that you get them right. In this section, I offer guidance on putting together the functions that comprise the backbone of well-formed classes.

Back to Item 10: Write operator delete if you write operator new.
Continue to Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.

Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.

Consider a class for representing `String` objects:

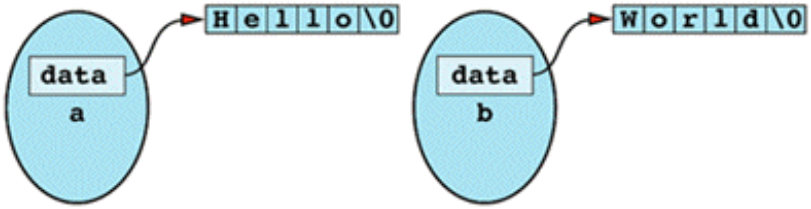
```
// a poorly designed String class
class String {
public:
  String(const char *value);
  ~String();

  ...                           // no copy ctor or operator=

private:
  char *data;
};

String::String(const char *value)
{
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  }
  else {
    data = new char[1];
    *data = '\0';
  }
}

inline String::~String() { delete [] data; }
```

Note that there is no assignment operator or copy constructor declared in this class. As you'll see, this has some unfortunate consequences.

If you make these object definitions,

```
String a("Hello");
String b("World");
```

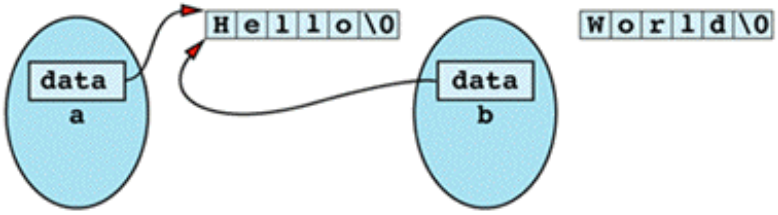
the situation is as shown below:



Inside object `a` is a pointer to memory containing the character string "Hello". Separate from that is an object `b` containing a pointer to the character string "World". If you now perform an assignment,

```
b = a;
```

there is no client-defined `operator=` to call, so C++ generates and calls the default assignment operator instead (see Item 45). This default assignment operator performs memberwise assignment from the members of `a` to the members of `b`, which for pointers (`a.data` and `b.data`) is just a bitwise copy. The result of this assignment is shown below.



There are at least two problems with this state of affairs. First, the memory that `b` used to point to was never deleted; it is lost forever. This is a classic example of how a memory leak can arise. Second, both `a` and `b` now contain pointers to the same character string. When one of them goes out of scope, its destructor will delete the memory still pointed to by the other. For example:

```
String a("Hello");                  // define and construct a


{                                   // open new scope
  String b("World");                // define and construct b
```

```
  ...


  b = a;                          // execute default op=,
                                  // lose b's memory


}                                 // close scope, call b's
                                  // destructor


String c = a;                     // c.data is undefined!
                                  // a.data is already deleted
```

The last statement in this example is a call to the copy constructor, which also isn't defined in the class, hence will be generated by C++ in the same manner as the assignment operator (again, see Item 45) and with the same behavior: bitwise copy of the underlying pointers. That leads to the same kind of problem, but without the worry of a memory leak, because the object being initialized can't yet point to any allocated memory. In the case of the code above, for example, there is no memory leak when `c.data` is initialized with the value of `a.data`, because `c.data` doesn't yet point anywhere. However, after `c` is initialized with `a`, both `c.data` and `a.data` point to the same place, so that place will be deleted twice: once when `c` is destroyed, once again when `a` is destroyed.

The case of the copy constructor differs a little from that of the assignment operator, however, because of the way it can bite you: pass-by-value. Of course, Item 22 demonstrates that you should only rarely pass objects by value, but consider this anyway:

```
void doNothing(String localString) {}


String s = "The Truth Is Out There";


doNothing(s);
```

Everything looks innocuous enough, but because `localString` is passed by value, it must be initialized from `s` via the (default) copy constructor. Hence, `localString` has a copy of the *pointer* that is inside `s`. When `doNothing` finishes executing, `localString` goes out of scope, and its destructor is called. The end result is by now familiar: `s` contains a pointer to memory that `localString` has already deleted.

By the way, the result of using `delete` on a pointer that has already been deleted is undefined, so even if `s` is never used again, there could well be a problem when it goes out of scope.

The solution to these kinds of pointer aliasing problems is to write your own versions of the copy constructor and the assignment operator if you have any pointers in your class. Inside those functions, you can either copy the pointed-to data structures so that every object has its own copy, or you can implement some kind of reference-counting scheme (see Item M29) to keep track of how many objects are currently pointing to a particular data structure. The reference-counting approach is more complicated, and it calls for extra work inside the constructors and destructors, too, but in some (though by no means all) applications, it can result in significant memory savings and substantial increases in speed.

For some classes, it's more trouble than it's worth to implement copy constructors and assignment operators, especially when you have reason to believe that your clients won't make copies or perform assignments. The examples above demonstrate that omitting the corresponding member functions reflects poor design, but what do you do if writing them isn't practical, either? Simple: you follow this Item's advice. You *declare* the functions (`private`, as it turns out), but you don't define (i.e., implement) them at all. That prevents clients from calling them, and it prevents compilers from generating them, too. For details on this nifty trick, see Item 27.

One more thing about the `String` class I used in this Item. In the constructor body, I was careful to use `[]` with `new` both times I called it, even though in one of the places I wanted only a single object. As described in Item 5, it's essential to employ the same form in corresponding applications of `new` and `delete`, so I was careful to be consistent in my uses of `new`. This is something you do *not* want to forget. *Always* make sure that you use `[]` with `delete` if and only if you used `[]` with the corresponding use of `new`.

Back to Constructors, Destructors, and Assignment Operators
Continue to Item 12: Prefer initialization to assignment in constructors.

Back to Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.
Continue to Item 13: List members in an initialization list in the order in which they are declared.

Item 12: Prefer initialization to assignment in constructors.

Consider a template for generating classes that allow a name to be associated with a pointer to an object of some type T:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...


private:
  string name;
  T *ptr;
};
```

(In light of the aliasing that can arise during the assignment and copy construction of objects with pointer members (see Item 11), you might wish to consider whether `NamedPtr` should implement these functions. Hint: it should (see Item 27).)

When you write the `NamedPtr` constructor, you have to transfer the values of the parameters to the corresponding data members. There are two ways to do this. The first is to use the member initialization list:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr  )
: name(initName), ptr(initPtr)
{}
```

The second is to make assignments in the constructor body:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
{
  name = initName;
  ptr = initPtr;
}
```

There are important differences between these two approaches.

From a purely pragmatic point of view, there are times when the initialization list *must* be used. In particular, `const` and reference members may *only* be initialized, never assigned. So, if you decided that a `NamedPtr<T>` object could never change its name or its pointer, you might follow the advice of Item 21 and declare the members `const`:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...


private:
  const string name;
  T * const ptr;
};
```

This class definition *requires* that you use a member initialization list, because `const` members may only be initialized, never assigned.

You'd obtain very different behavior if you decided that a `NamedPtr<T>` object should contain a *reference* to an existing name. Even so, you'd still have to initialize the reference on your constructors' member initialization lists. Of course, you could also combine the two, yielding `NamedPtr<T>` objects with read-only access to names that might be modified outside the class:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...


private:
  const string& name;               // must be initialized via
                                    // initializer list



  T * const ptr;                    // must be initialized via
                                    // initializer list
};
```

The original class template, however, contains no `const` or reference members. Even so, using a member initialization list is still preferable to performing assignments inside the constructor. This

time the reason is efficiency. When a member initialization list is used, only a single `string` member function is called. When assignment inside the constructor is used, two are called. To understand why, consider what happens when you declare a `NamedPtr<T>` object.

Construction of objects proceeds in two phases:

1. Initialization of data members. (See also Item 13.)
2. Execution of the body of the constructor that was called.

(For objects with base classes, base class member initialization and constructor body execution occurs prior to that for derived classes.)

For the `NamedPtr` classes, this means that a constructor for the `string` object `name` will *always* be called before you ever get inside the body of a `NamedPtr` constructor. The only question, then, is this: which `string` constructor will be called?

That depends on the member initialization list in the `NamedPtr` classes. If you fail to specify an initialization argument for `name`, the default `string` constructor will be called. When you later perform an assignment to `name` inside the `NamedPtr` constructors, you will call `operator=` on `name`. That will total two calls to `string` member functions: one for the default constructor and one more for the assignment.

On the other hand, if you use a member initialization list to specify that `name` should be initialized with `initName`, `name` will be initialized through the copy constructor at a cost of only a single function call.

Even in the case of the lowly `string` type, the cost of an unnecessary function call may be significant, and as classes become larger and more complex, so do their constructors, and so does the cost of constructing objects. If you establish the habit of using a member initialization list whenever you can, not only do you satisfy a requirement for `const` and reference members, you also minimize the chances of initializing data members in an inefficient manner.

In other words, initialization via a member initialization list is *always* legal, is *never* less efficient than assignment inside the body of the constructor, and is often *more* efficient. Furthermore, it simplifies maintenance of the class (see Item M32), because if a data member's type is later modified to something that *requires* use of a member initialization list, nothing has to change.

There is one time, however, when it may make sense to use assignment instead of initialization for the data members in a class. That is when you have a large number of data members of *built-in types*, and you want them all initialized the same way in each constructor. For example, here's a class that might qualify for this kind of treatment:

```
class ManyDataMbrs {
public:
  // default constructor
  ManyDataMbrs();


  // copy constructor
  ManyDataMbrs(const ManyDataMbrs& x);


private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;
};
```

Suppose you want to initialize all the ints to 1 and all the doubles to 0, even if the copy constructor is used. Using member initialization lists, you'd have to write this:

```
ManyDataMbrs::ManyDataMbrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }


ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

This is more than just unpleasant drudge work. It is error-prone in the short term and difficult to maintain in the long term.

However, you can take advantage of the fact that there is no operational difference between initialization and assignment for (non-`const`, non-reference) objects of built-in types, so you can safely replace the memberwise initialization lists with a function call to a common initialization routine:

```
class ManyDataMbrs {
public:
  // default constructor
  ManyDataMbrs();


  // copy constructor
```

```
  ManyDataMbrs(const ManyDataMbrs& x);


private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;



  void init();                  // used to initialize data
                                // members
};


void ManyDataMbrs::init()
{
  a = b = c = d = e = f = g = h = 1;
  i = j = k = l = m = 0;
}


ManyDataMbrs::ManyDataMbrs()
{
  init();


  ...


}


ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
{
  init();


  ...


}
```

Because the initialization routine is an implementation detail of the class, you are, of course, careful to make it `private`, right?

Note that `static` class members should *never* be initialized in a class's constructor. Static members

are initialized only once per program run, so it makes no sense to try to "initialize" them each time an object of the class's type is created. At the very least, doing so would be inefficient: why pay to "initialize" an object multiple times? Besides, initialization of static class members is different enough from initialization of their nonstatic counterparts that an entire Item — Item 47 — is devoted to the topic.

Back to Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.
Continue to Item 13: List members in an initialization list in the order in which they are declared.

Back to Item 12: Prefer initialization to assignment in constructors.
Continue to Item 14: Make sure base classes have virtual destructors.

Item 13: List members in an initialization list in the order in which they are declared.

Unrepentant Pascal and Ada programmers often yearn for the ability to define arrays with arbitrary bounds, i.e., from 10 to 20 instead of from 0 to 10. Long-time C programmers will insist that everybody who's anybody will always start counting from 0, but it's easy enough to placate the `begin`/`end` crowd. All you have to do is define your own `Array` class template:

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  ...


private:
  vector<T> data;               // the array data is stored
                                // in a vector object; see
                                // Item 49 for info about
                                // the vector template


  size_t size;                  // # of elements in array


  int lBound, hBound;           // lower bound, higher bound
};


template<class T>
Array<T>::Array(int lowBound, int highBound)
: size(highBound - lowBound + 1),
  lBound(lowBound), hBound(highBound),
  data(size)
{}
```

An industrial-strength constructor would perform sanity checking on its parameters to ensure that `highBound` was at least as great as `lowBound`, but there is a much nastier error here: even with perfectly good values for the array's bounds, you have absolutely no idea how many elements `data` holds.

"How can that be?" I hear you cry. "I carefully initialized `size` before passing it to the `vector` constructor!" Unfortunately, you didn't — you just tried to. The rules of the game are that class members are initialized *in the order of their declaration in the class*; the order in which they are listed in a member initialization list makes not a whit of difference. In the classes generated by your `Array` template, `data` will always be initialized first, followed by `size`, `lBound`, and `hBound`. Always.

Perverse though this may seem, there is a reason for it. Consider this scenario:

```
class Wacko {
public:
  Wacko(const char *s): s1(s), s2(0) {}
  Wacko(const Wacko& rhs): s2(rhs.s1), s1(0) {}


private:
  string s1, s2;
};


Wacko w1 = "Hello world!";
Wacko w2 = w1;
```

If members were initialized in the order of their appearance in an initialization list, the data members of `w1` and `w2` would be constructed in different orders. Recall that the destructors for the members of an object are always called in the inverse order of their constructors. Thus, if the above were allowed, compilers would have to keep track of the order in which the members were initialized for *each object*, just to ensure that the destructors would be called in the right order. That would be an expensive proposition. To avoid that overhead, the order of construction and destruction is the same for all objects of a given type, and the order of members in an initialization list is ignored.

Actually, if you really want to get picky about it, only nonstatic data members are initialized according to the rule. Static data members act like global and namespace objects, so they are initialized only once; see Item 47 for details. Furthermore, base class data members are initialized before derived class data members, so if you're using inheritance, you should list base class initializers at the very beginning of your member initialization lists. (If you're using *multiple* inheritance, your base classes will be initialized in the order in which you *inherit* from them; the order in which they're listed in your member initialization lists will again be ignored. However, if you're using multiple inheritance, you've probably got more important things to worry about. If you don't, Item 43 would be happy to make suggestions regarding aspects of multiple inheritance that are worrisome.)

The bottom line is this: if you hope to understand what is really going on when your objects are initialized, be sure to list the members in an initialization list in the order in which those members are declared in the class.

Back to Item 12: Prefer initialization to assignment in constructors.
Continue to Item 14: Make sure base classes have virtual destructors.

Back to Item 13: List members in an initialization list in the order in which they are declared.
Continue to Item 15: Have operator= return a reference to *this.

Item 14: Make sure base classes have virtual destructors.

Sometimes it's convenient for a class to keep track of how many objects of its type exist. The straightforward way to do this is to create a static class member for counting the objects. The member is initialized to 0, is incremented in the class constructors, and is decremented in the class destructor. (Item M26 shows how to package this approach so it's easy to add to any class, and my article on counting objects describes additional refinements to the technique.)

You might envision a military application, in which a class representing enemy targets might look something like this:

```
class EnemyTarget {
public:
  EnemyTarget() { ++numTargets; }
  EnemyTarget(const EnemyTarget&) { ++numTargets; }
  ~EnemyTarget() { --numTargets; }


  static size_t numberOfTargets()
  { return numTargets; }



  virtual bool destroy();                 // returns success of
                                          // attempt to destroy
                                          // EnemyTarget object


private:
  static size_t numTargets;               // object counter
};


// class statics must be defined outside the class;
// initialization is to 0 by default
size_t EnemyTarget::numTargets;
```

This class is unlikely to win you a government defense contract, but it will suffice for our purposes here, which are substantially less demanding than are those of the Department of Defense. Or so we may hope.

Let us suppose that a particular kind of enemy target is an enemy tank, which you model, naturally enough (see Item 35, but also see Item M33), as a publicly derived class of `EnemyTarget`. Because

you are interested in the total number of enemy tanks as well as the total number of enemy targets, you'll pull the same trick with the derived class that you did with the base class:

```
class EnemyTank: public EnemyTarget {
public:
  EnemyTank() { ++numTanks; }


  EnemyTank(const EnemyTank& rhs)
  : EnemyTarget(rhs)
  { ++numTanks; }


  ~EnemyTank() { --numTanks; }


  static size_t numberOfTanks()
  { return numTanks; }


  virtual bool destroy();


private:
  static size_t numTanks;         // object counter for tanks
};
```

Having now added this code to two different classes, you may be in a better position to appreciate Item M26's general solution to the problem.

Finally, let's assume that somewhere in your application, you dynamically create an `EnemyTank` object using `new`, which you later get rid of via `delete`:

```
EnemyTarget *targetPtr = new EnemyTank;


...


delete targetPtr;
```

Everything you've done so far seems completely kosher. Both classes undo in the destructor what they did in the constructor, and there's certainly nothing wrong with your application, in which you were careful to use `delete` after you were done with the object you conjured up with `new`. Nevertheless, there is something very troubling here. Your program's behavior is *undefined* — you

have no way of knowing what will happen.

The C++ language standard is unusually clear on this topic. When you try to delete a derived class object through a base class pointer and the base class has a nonvirtual destructor (as `EnemyTarget` does), the results are undefined. That means compilers may generate code to do whatever they like: reformat your disk, send suggestive mail to your boss, fax source code to your competitors, whatever. (What often happens at runtime is that the derived class's destructor is never called. In this example, that would mean your count of `EnemyTanks` would not be adjusted when `targetPtr` was deleted. Your count of enemy tanks would thus be wrong, a rather disturbing prospect to combatants dependent on accurate battlefield information.)

To avoid this problem, you have only to make the `EnemyTarget` destructor *virtual*. Declaring the destructor virtual ensures well-defined behavior that does precisely what you want: both `EnemyTank`'s and `EnemyTarget`'s destructors will be called before the memory holding the object is deallocated.

Now, the `EnemyTarget` class contains a virtual function, which is generally the case with base classes. After all, the purpose of virtual functions is to allow customization of behavior in derived classes (see Item 36), so almost all base classes contain virtual functions.

If a class does *not* contain any virtual functions, that is often an indication that it is not meant to be used as a base class. When a class is not intended to be used as a base class, making the destructor virtual is usually a bad idea. Consider this example, based on a discussion in the ARM (see Item 50):

```
// class for representing 2D points
class Point {
public:
  Point(short int xCoord, short int yCoord);
  ~Point();


private:
  short int x, y;
};
```

If a `short int` occupies 16 bits, a `Point` object can fit into a 32-bit register. Furthermore, a `Point` object can be passed as a 32-bit quantity to functions written in other languages such as C or FORTRAN. If `Point`'s destructor is made virtual, however, the situation changes.

The implementation of virtual functions requires that objects carry around with them some additional information that can be used at runtime to determine which virtual functions should be invoked on the object. In most compilers, this extra information takes the form of a pointer called a `vptr` ("virtual table pointer"). The `vptr` points to an array of function pointers called a `vtbl` ("virtual table"); each class with virtual functions has an associated `vtbl`. When a virtual function is invoked on an object, the actual function called is determined by following the object's `vptr` to a

vtbl and then looking up the appropriate function pointer in the vtbl.

The details of how virtual functions are implemented are unimportant (though, if you're curious, you can read about them in Item M24). What *is* important is that if the Point class contains a virtual function, objects of that type will implicitly *double* in size, from two 16-bit shorts to two 16-bit shorts plus a 32-bit vptr! No longer will Point objects fit in a 32-bit register. Furthermore, Point objects in C++ no longer look like the same structure declared in another language such as C, because their foreign language counterparts will lack the vptr. As a result, it is no longer possible to pass Points to and from functions written in other languages unless you explicitly compensate for the vptr, which is itself an implementation detail and hence unportable.

The bottom line is that gratuitously declaring all destructors virtual is just as wrong as never declaring them virtual. In fact, many people summarize the situation this way: declare a virtual destructor in a class if and only if that class contains at least one virtual function.

This is a good rule, one that works most of the time, but unfortunately, it is possible to get bitten by the nonvirtual destructor problem even in the absence of virtual functions. For example, Item 13 considers a class template for implementing arrays with client-defined bounds. Suppose you decide (in spite of the advice in Item M33) to write a template for derived classes representing named arrays, i.e., classes where every array has a name:

```
template<class T>               // base class template
class Array {                   // (from Item 13)
public:
  Array(int lowBound, int highBound);
  ~Array();


private:
  vector<T> data;
  size_t size;
  int lBound, hBound;
};


template<class T>
class NamedArray: public Array<T> {
public:
  NamedArray(int lowBound, int highBound, const string& name);
  ...


private:
  string arrayName;
};
```

If anywhere in an application you somehow convert a pointer-to-NamedArray into a pointer-to-

Array and you then use delete on the Array pointer, you are instantly transported to the realm of undefined behavior:

```
NamedArray<int> *pna =
  new NamedArray<int>(10, 20, "Impending Doom");


Array<int> *pa;


...


pa = pna;               // NamedArray<int>* -> Array<int>*


...


delete pa;              // undefined! (Insert theme to
                        //◦Twilight Zone here); in practice,
                        // pa->arrayName will often be leaked,
                        // because the NamedArray part of
                        // *pa will never be destroyed
```

This situation can arise more frequently than you might imagine, because it's not uncommon to want to take an existing class that does something, Array in this case, and derive from it a class that does all the same things, plus more. NamedArray doesn't redefine any of the behavior of Array — it inherits all its functions without change — it just adds some additional capabilities. Yet the nonvirtual destructor problem persists. (As do others. See Item M33.)

Finally, it's worth mentioning that it can be convenient to declare pure virtual destructors in some classes. Recall that pure virtual functions result in *abstract* classes — classes that can't be instantiated (i.e., you can't create objects of that type). Sometimes, however, you have a class that you'd like to be abstract, but you don't happen to have any functions that are pure virtual. What to do? Well, because an abstract class is intended to be used as a base class, and because a base class should have a virtual destructor, and because a pure virtual function yields an abstract class, the solution is simple: declare a pure virtual destructor in the class you want to be abstract.

Here's an example:

```
class AWOV {                // AWOV = "Abstract w/o
                            // Virtuals"
```

```
public:
  virtual ~AWOV() = 0;      // declare pure virtual
                            // destructor
};
```

This class has a pure virtual function, so it's abstract, and it has a virtual destructor, so you can rest assured that you won't have to worry about the destructor problem. There is one twist, however: you must provide a *definition* for the pure virtual destructor:

```
AWOV::~AWOV() {}           // definition of pure
                           // virtual destructor
```

You need this definition, because the way virtual destructors work is that the most derived class's destructor is called first, then the destructor of each base class is called. That means that compilers will generate a call to `~AWOV` even though the class is abstract, so you have to be sure to provide a body for the function. If you don't, the linker will complain about a missing symbol, and you'll have to go back and add one.

You can do anything you like in that function, but, as in the example above, it's not uncommon to have nothing to do. If that is the case, you'll probably be tempted to avoid paying the overhead cost of a call to an empty function by declaring your destructor inline. That's a perfectly sensible strategy, but there's a twist you should know about.

Because your destructor is virtual, its address must be entered into the class's `vtbl` (see Item M24). But inline functions aren't supposed to exist as freestanding functions (that's what `inline` means, right?), so special measures must be taken to get addresses for them. Item 33 tells the full story, but the bottom line is this: if you declare a virtual destructor `inline`, you're likely to avoid function call overhead when it's invoked, but your compiler will still have to generate an out-of-line copy of the function somewhere, too.

Back to Item 13: List members in an initialization list in the order in which they are declared.
Continue to Item 15: Have operator= return a reference to *this.

Back to Item 14: Make sure base classes have virtual destructors.
Continue to Item 16: Assign to all data members in operator=.

Item 15: Have `operator=` return a reference to `*this`.

◦Bjarne Stroustrup, the designer of C++, went to a lot of trouble to ensure that user-defined types would mimic the built-in types as closely as possible. That's why you can overload operators, write type conversion functions (see Item M5), take control of assignment and copy construction, etc. After so much effort on his part, the least you can do is keep the ball rolling.

Which brings us to assignment. With the built-in types, you can chain assignments together, like so:

```
int w, x, y, z;


w = x = y = z = 0;
```

As a result, you should be able to chain together assignments for user-defined types, too:

```
string w, x, y, z;              // string is "user-defined"
                                // by the standard C++
                                // library (see Item 49)


w = x = y = z = "Hello";
```

As fate would have it, the assignment operator is right-associative, so the assignment chain is parsed like this:

```
w = (x = (y = (z = "Hello")));
```

It's worthwhile to write this in its completely equivalent functional form. Unless you're a closet LISP programmer, this example should make you grateful for the ability to define infix operators:

```
w.operator=(x.operator=(y.operator=(z.operator=("Hello"))));
```

This form is illustrative because it emphasizes that the argument to `w.operator=`, `x.operator=`,

and `y.operator=` is the return value of a previous call to `operator=`. As a result, the return type of `operator=` must be acceptable as an input to the function itself. For the default version of `operator=` in a class `C`, the signature of the function is as follows (see Item 45):

```
C& C::operator=(const C&);
```

You'll almost always want to follow this convention of having `operator=` both take and return a reference to a class object, although at times you may overload `operator=` so that it takes different argument types. For example, the standard `string` type provides two different versions of the assignment operator:

```
string&                             // assign a string
operator=(const string& rhs);       // to a string



string&                             // assign a char*
operator=(const char *rhs);         // to a string
```

Notice, however, that even in the presence of overloading, the return type is a reference to an object of the class.

A common error amongst new C++ programmers is to have `operator=` return `void`, a decision that seems reasonable until you realize it prevents chains of assignment. So don't do it.

Another common error is to have `operator=` return a reference to a `const` object, like this:

```
class Widget {
public:
  ...                                                 // note
  const Widget& operator=(const Widget& rhs);         // const
  ...                                                 // return
};                                                    // type
```

The usual motivation is to prevent clients from doing silly things like this:

```
Widget w1, w2, w3;



...
```

```
(w1 = w2) = w3;          // assign w2 to w1, then w3 to
                         // the result! (Giving Widget's
                         // operator= a const return value
                         // prevents this from compiling.)
```

Silly this may be, but not so silly that it's prohibited for the built-in types:

```
int i1, i2, i3;


...



(i1 = i2) = i3;                // legal! assigns i2 to
                               // i1, then i3 to i1!
```

I know of no practical use for this kind of thing, but if it's good enough for the `int`s, it's good enough for me and my classes. It should be good enough for you and yours, too. Why introduce gratuitous incompatibilities with the conventions followed by the built-in types?

Within an assignment operator bearing the default signature, there are two obvious candidates for the object to be returned: the object on the left hand side of the assignment (the one pointed to by `this`) and the object on the right-hand side (the one named in the parameter list). Which is correct?

Here are the possibilities for a `String` class (a class for which you'd definitely want to write an assignment operator, as explained in Item 11):

```
String& String::operator=(const String& rhs)
{


  ...



  return *this;            // return reference
                           // to left-hand object
}
```

```
String& String::operator=(const String& rhs)
{

  ...


  return rhs;             // return reference to
                          // right-hand object
}
```

This might strike you as a case of six of one versus a half a dozen of the other, but there are important differences.

First, the version returning `rhs` won't compile. That's because `rhs` is a reference-to-*`const`*-`String`, but `operator=` returns a reference-to-`String`. Compilers will give you no end of grief for trying to return a reference-to-non-`const` when the object itself is `const`. That seems easy enough to get around, however — just redeclare `operator=` like this:

```
String& String::operator=(String& rhs)   { ... }
```

Alas, now the client code won't compile! Look again at the last part of the original chain of assignments:

```
x = "Hello";                    // same as x.op=("Hello");
```

Because the right-hand argument of the assignment is not of the correct type — it's a `char` array, not a `String` — compilers would have to create a temporary `String` object (via the `String` constructor — see Item M19) to make the call succeed. That is, they'd have to generate code roughly equivalent to this:

```
const String temp("Hello");     // create temporary


x = temp;                       // pass temporary to op=
```

Compilers are willing to create such a temporary (unless the needed constructor is `explicit` — see Item 19), but note that the temporary object is `const`. This is important, because it prevents you from accidentally passing a temporary into a function that modifies its parameter. If that were

allowed, programmers would be surprised to find that only the compiler-generated temporary was modified, not the argument they actually provided at the call site. (We know this for a fact, because early versions of C++ allowed these kinds of temporaries to be generated, passed, and modified, and the result was a lot of surprised programmers.)

Now we can see why the client code above won't compile if `String`'s `operator=` is declared to take a reference-to-non-`const` `String`: it's never legal to pass a `const` object to a function that fails to declare the corresponding parameter `const`. That's just simple `const`-correctness.

You thus find yourself in the happy circumstance of having no choice whatsoever: you'll always want to define your assignment operators in such a way that they return a reference to their left-hand argument, `*this`. If you do anything else, you prevent chains of assignments, you prevent implicit type conversions at call sites, or both.

Back to Item 14: Make sure base classes have virtual destructors.
Continue to Item 16: Assign to all data members in operator=.

Back to Item 15: Have operator= return a reference to *this.
Continue to Item 17: Check for assignment to self in operator=.

Item 16: Assign to all data members in `operator=`.

Item 45 explains that C++ will write an assignment operator for you if you don't declare one yourself, and Item 11 describes why you often won't much care for the one it writes for you, so perhaps you're wondering if you can somehow have the best of both worlds, whereby you let C++ generate a default assignment operator and you selectively override those parts you don't like. No such luck. If you want to take control of any part of the assignment process, you must do the entire thing yourself.

In practice, this means that you need to assign to *every* data member of your object when you write your assignment operator(s):

```
template<class T>          // template for classes associating
class NamedPtr {           // names with pointers (from Item 12)
public:
  NamedPtr(const string& initName, T *initPtr);
  NamedPtr& operator=(const NamedPtr& rhs);


private:
  string name;
  T *ptr;
};


template<class T>
NamedPtr<T>& NamedPtr<T>::operator=(const NamedPtr<T>& rhs)
{
  if (this == &rhs)
    return *this;              // see Item 17


  // assign to all data members
  name = rhs.name;             // assign to name



  *ptr = *rhs.ptr;             // for ptr, assign what's
                               // pointed to, not the
                               // pointer itself
```

```
  return *this;                 // see Item 15
}
```

This is easy enough to remember when the class is originally written, but it's equally important that the assignment operator(s) be updated if new data members are added to the class. For example, if you decide to upgrade the `NamedPtr` template to carry a timestamp marking when the name was last changed, you'll have to add a new data member, and this will require updating the constructor(s) as well as the assignment operator(s). In the hustle and bustle of upgrading a class and adding new member functions, etc., it's easy to let this kind of thing slip your mind.

The real fun begins when inheritance joins the party, because a derived class's assignment operator(s) must also handle assignment of its base class members! Consider this:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}


private:
  int x;
};


class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}


  Derived& operator=(const Derived& rhs);


private:
  int y;
};
```

The logical way to write `Derived`'s assignment operator is like this:

```
// erroneous assignment operator
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;    // see Item 17
```

```
  y = rhs.y;                          // assign to Derived's
                                      // lone data member



  return *this;                       // see Item 15
}
```

Unfortunately, this is incorrect, because the data member `x` in the `Base` part of a `Derived` object is unaffected by this assignment operator. For example, consider this code fragment:

```
void assignmentTester()
{
  Derived d1(0);                       // d1.x = 0, d1.y = 0
  Derived d2(1);                       // d2.x = 1, d2.y = 1



  d1 = d2;         // d1.x = 0, d1.y = 1!
}
```

Notice how the `Base` part of `d1` is unchanged by the assignment.

The straightforward way to fix this problem would be to make an assignment to `x` in `Derived::operator=`. Unfortunately, that's not legal, because `x` is a private member of `Base`. Instead, you have to make an explicit assignment to the `Base` *part* of `Derived` from inside `Derived`'s assignment operator.

This is how you do it:

```
// correct assignment operator
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;



  Base::operator=(rhs);    // call this->Base::operator=
  y = rhs.y;



  return *this;
}
```

Here you just make an explicit call to `Base::operator=`. That call, like all calls to member functions from within other member functions, will use `*this` as its implicit left-hand object. The

result will be that `Base::operator=` will do whatever work it does on the `Base` part of `*this` — precisely the effect you want.

Alas, some compilers (incorrectly) reject this kind of call to a base class's assignment operator if that assignment operator was generated by the compiler (see Item 45). To pacify these renegade translators, you need to implement `Derived::operator=` this way:

```
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;



  static_cast<Base&>(*this) = rhs;      // call operator= on
                                        // Base part of *this
  y = rhs.y;



  return *this;
}
```

This monstrosity casts `*this` to be a reference to a `Base`, then makes an assignment to the result of the cast. That makes an assignment to only the `Base` part of the `Derived` object. Careful now! It is important that the cast be to a *reference* to a `Base` object, not to a `Base` object itself. If you cast `*this` to be a `Base` object, you'll end up calling the copy constructor for `Base`, and the new object you construct (see Item M19) will be the target of the assignment; `*this` will remain unchanged. Hardly what you want.

Regardless of which of these approaches you employ, once you've assigned the `Base` part of the `Derived` object, you then continue with `Derived`'s assignment operator, making assignments to all the data members of `Derived`.

A similar inheritance-related problem often arises when implementing derived class copy constructors. Take a look at the following, which is the copy constructor analogue of the code we just examined:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}
  Base(const Base& rhs): x(rhs.x) {}



private:
  int x;
```

```
};


class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}



  Derived(const Derived& rhs)      // erroneous copy
  : y(rhs.y) {}                    // constructor


private:
  int y;
};
```

Class `Derived` demonstrates one of the nastiest bugs in all C++-dom: it fails to copy the base class part when a `Derived` object is copy constructed. Of course, the `Base` part of such a `Derived` object is constructed, but it's constructed using `Base`'s *default* constructor. Its member `x` is initialized to 0 (the default constructor's default parameter value), regardless of the value of `x` in the object being copied!

To avoid this problem, `Derived`'s copy constructor must make sure that `Base`'s copy constructor is invoked instead of `Base`'s default constructor. That's easily done. Just be sure to specify an initializer value for `Base` in the member initialization list of `Derived`'s copy constructor:

```
class Derived: public Base {
public:
  Derived(const Derived& rhs): Base(rhs), y(rhs.y) {}


  ...


};
```

Now when a client creates a `Derived` by copying an existing object of that type, its `Base` part will be copied, too.

Back to Item 15: Have operator= return a reference to *this.
Continue to Item 17: Check for assignment to self in operator=.

Back to Item 16: Assign to all data members in operator=.
Continue to Classes and Functions: Design and Declaration

Item 17: Check for assignment to self in `operator=`.

An assignment to self occurs when you do something like this:

```
class X { ... };

X a;

a = a;                    // a is assigned to itself
```

This looks like a silly thing to do, but it's perfectly legal, so don't doubt for a moment that programmers do it. More importantly, assignment to self can appear in this more benign-looking form:

```
a = b;
```

If `b` is another name for `a` (for example, a reference that has been initialized to `a`), then this is also an assignment to self, though it doesn't outwardly look like it. This is an example of *aliasing*: having two or more names for the same underlying object. As you'll see at the end of this Item, aliasing can crop up in any number of nefarious disguises, so you need to take it into account any time you write a function.

Two good reasons exist for taking special care to cope with possible aliasing in assignment operator(s). The lesser of them is efficiency. If you can detect an assignment to self at the top of your assignment operator(s), you can return right away, possibly saving a lot of work that you'd otherwise have to go through to implement assignment. For example, Item 16 points out that a proper assignment operator in a derived class must call an assignment operator for each of its base classes, and those classes might themselves be derived classes, so skipping the body of an assignment operator in a derived class might save a large number of other function calls.

A more important reason for checking for assignment to self is to ensure correctness. Remember that an assignment operator must typically free the resources allocated to an object (i.e., get rid of its old value) before it can allocate the new resources corresponding to its new value. When assigning to self, this freeing of resources can be disastrous, because the old resources might be needed during the process of allocating the new ones.

Consider assignment of `String` objects, where the assignment operator fails to check for assignment to self:

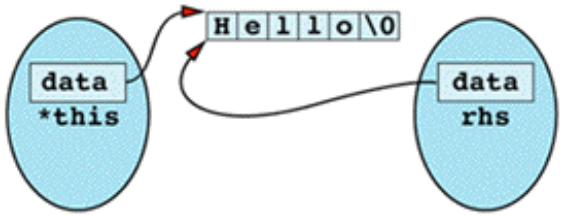
```
class String {
public:
  String(const char *value);    // see Item 11 for
                                // function definition



  ~String();                    // see Item 11 for
                                // function definition
  ...


  String& operator=(const String& rhs);



private:
  char *data;
};



// an assignment operator that omits a check
// for assignment to self
String& String::operator=(const String& rhs)
{
  delete [] data;    // delete old memory



  // allocate new memory and copy rhs's value into it
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);



  return *this;      // see Item 15
}
```

Consider now what happens in this case:

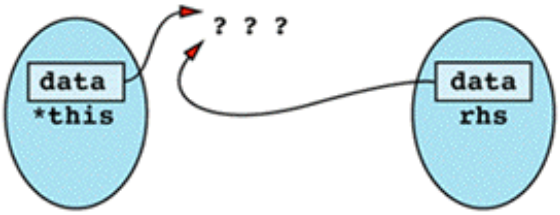
```
String a = "Hello";



a = a;               // same as a.operator=(a)
```

Inside the assignment operator, `*this` and `rhs` seem to be different objects, but in this case they

happen to be different names for the same object. You can envision it like this:



The first thing the assignment operator does is use `delete` on `data`, and the result is the following state of affairs:



Now when the assignment operator tries to do a `strlen` on `rhs.data`, the results are undefined. This is because `rhs.data` was deleted when `data` was deleted, which happened because `data`, `this->data`, and `rhs.data` are all the same pointer! From this point on, things can only get worse.

By now you know that the solution to the dilemma is to check for an assignment to self and to return immediately if such an assignment is detected. Unfortunately, it's easier to talk about such a check than it is to write it, because you are immediately forced to figure out what it means for two objects to be "the same."

The topic you confront is technically known as that of *object identity*, and it's a well-known topic in object-oriented circles. This book is no place for a discourse on object identity, but it is worthwhile to mention the two basic approaches to the problem.

One approach is to say that two objects are the same (have the same identity) if they have the same value. For example, two `String` objects would be the same if they represented the same sequence of characters:

```
String a = "Hello";
String b = "World";
```

```
String c = "Hello";
```

Here `a` and `c` have the same value, so they are considered identical; `b` is different from both of them. If you wanted to use this definition of identity in your `String` class, your assignment operator might look like this:

```
String& String::operator=(const String& rhs)
{
  if (strcmp(data, rhs.data) == 0) return *this;

  ...

}
```

Value equality is usually determined by `operator==`, so the general form for an assignment operator for a class `C` that uses value equality for object identity is this:

```
C& C::operator=(const C& rhs)
{
  // check for assignment to self
  if (*this == rhs)              // assumes op== exists
    return *this;

  ...

}
```

Note that this function is comparing *objects* (via `operator==`), not pointers. Using value equality to determine identity, it doesn't matter whether two objects occupy the same memory; all that matters is the values they represent.

The other possibility is to equate an object's identity with its address in memory. Using this definition of object equality, two objects are the same if and only if they have the same address. This definition is more common in C++ programs, probably because it's easy to implement and the computation is fast, neither of which is always true when object identity is based on values. Using address equality, a general assignment operator looks like this:

```
C& C::operator=(const C& rhs)
{
  // check for assignment to self
  if (this == &rhs) return *this;


  ...


}
```

This suffices for a great many programs.

If you need a more sophisticated mechanism for determining whether two objects are the same, you'll have to implement it yourself. The most common approach is based on a member function that returns some kind of object identifier:

```
class C {
public:
  ObjectID identity() const;            // see also Item 36


  ...


};
```

Given object pointers `a` and `b`, then, the objects they point to are identical if and only if `a->identity() == b->identity()`. Of course, you are responsible for writing `operator==` for `ObjectID`s.

The problems of aliasing and object identity are hardly confined to `operator=`. That's just a function in which you are particularly likely to run into them. In the presence of references and pointers, any two names for objects of compatible types may in fact refer to the same object. Here are some other situations in which aliasing can show its Medusa-like visage:

```
class Base {
  void mf1(Base& rb);                       // rb and *this could be
                                            // the same
  ...


};
```

```
void f1(Base& rb1,Base& rb2);              // rb1 and rb2 could be
                                           // the same



class Derived: public Base {
  void mf2(Base& rb);                      // rb and *this could be
                                           // the same
  ...


};



int f2(Derived& rd, Base& rb);             // rd and rb could be
                                           // the same
```

These examples happen to use references, but pointers would serve just as well.

As you can see, aliasing can crop up in a variety of guises, so you can't just forget about it and hope you'll never run into it. Well, maybe *you* can, but most of us can't. At the expense of mixing my metaphors, this is a clear case in which an ounce of prevention is worth its weight in gold. Anytime you write a function in which aliasing could conceivably be present, you *must* take that possibility into account when you write the code.

Back to Item 16: Assign to all data members in operator=.
Continue to Classes and Functions: Design and Declaration

Back to Item 17: Check for assignment to self in operator=.
Continue to Item 18: Strive for class interfaces that are complete and minimal.

Classes and Functions: Design and Declaration

Declaring a new class in a program creates a new type: class design is *type* design. You probably don't have much experience with type design, because most languages don't offer you the opportunity to get any practice. In C++, it is of fundamental importance, not just because you can do it if you want to, but because you *are* doing it every time you declare a class, whether you mean to or not.

Designing good classes is challenging because designing good types is challenging. Good types have a natural syntax, an intuitive semantics, and one or more efficient implementations. In C++, a poorly thought out class definition can make it impossible to achieve any of these goals. Even the performance characteristics of a class's member functions are determined as much by the declarations of those member functions as they are by their definitions.

How, then, do you go about designing effective classes? First, you must understand the issues you face. Virtually every class requires that you confront the following questions, the answers to which often lead to constraints on your design:

- *How should objects be created and destroyed?* How this is done strongly influences the design of your constructors and destructor, as well as your versions of `operator new`, `operator new[]`, `operator delete`, and `operator delete[]`, if you write them. (Item M8 describes the differences among these terms.)
- *How does object initialization differ from object assignment?* The answer to this question determines the behavior of and the differences between your constructors and your assignment operators.
- *What does it mean to pass objects of the new type by value?* Remember, the copy constructor defines what it means to pass an object by value.
- *What are the constraints on legal values for the new type?* These constraints determine the kind of error checking you'll have to do inside your member functions, especially your constructors and assignment operators. It may also affect the exceptions your functions throw and, if you use them, your functions' exception specifications (see Item M14).
- *Does the new type fit into an inheritance graph?* If you inherit from existing classes, you are constrained by the design of those classes, particularly by whether the functions you inherit are virtual or nonvirtual. If you wish to allow other classes to inherit from your class, that will affect whether the functions you declare are virtual.
- *What kind of type conversions are allowed?* If you wish to allow objects of type A to be *implicitly* converted into objects of type B, you will want to write either a type conversion function in class A or a non-`explicit` constructor in class B that can be called with a single argument. If you wish to allow *explicit* conversions only, you'll want to write functions to perform the conversions, but you'll want to avoid making them type conversion operators or

non-`explicit` single-argument constructors. (Item M5 discusses the advantages and disadvantages of user-defined conversion functions.)

- *What operators and functions make sense for the new type?* The answer to this question determines which functions you'll declare in your class interface.
- *What standard operators and functions should be explicitly disallowed?* Those are the ones you'll need to declare `private`.
- *Who should have access to the members of the new type?* This question helps you determine which members are public, which are protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.
- How general is the new type? Perhaps you're not really defining a new type. Perhaps you're defining a whole *family* of types. If so, you don't want to define a new class, you want to define a new class *template*.

These are difficult questions to answer, so defining effective classes in C++ is far from simple. Done properly, however, user-defined classes in C++ yield types that are all but indistinguishable from built-in types, and that makes all the effort worthwhile.

A discussion of the details of each of the above issues would comprise a book in its own right, so the guidelines that follow are anything but comprehensive. However, they highlight some of the most important design considerations, warn about some of the most frequent errors, and provide solutions to some of the most common problems encountered by class designers. Much of the advice is as applicable to non-member functions as it is to member functions, so in this section I consider the design and declaration of global and namespace-resident functions, too.

Back to Item 17: Check for assignment to self in operator=.
Continue to Item 18: Strive for class interfaces that are complete and minimal.

Item 18: Strive for class interfaces that are complete and minimal.

The client interface for a class is the interface that is accessible to the programmers who use the class. Typically, only functions exist in this interface, because having data members in the client interface has a number of drawbacks (see Item 20).

Trying to figure out what functions should be in a class interface can drive you crazy. You're pulled in two completely different directions. On the one hand, you'd like to build a class that is easy to understand, straightforward to use, and easy to implement. That usually implies a fairly small number of member functions, each of which performs a distinct task. On other hand, you'd like your class to be powerful and convenient to use, which often means adding functions to provide support for commonly performed tasks. How do you decide which functions go into the class and which ones don't?

Try this: aim for a class interface that is *complete* and *minimal*.

A *complete* interface is one that allows clients to do anything they might reasonably want to do. That is, for any reasonable task that clients might want to accomplish, there is a reasonable way to accomplish it, although it may not be as convenient as clients might like. A *minimal* interface, on the other hand, is one with as few functions in it as possible, one in which no two member functions have overlapping functionality. If you offer a complete, minimal interface, clients can do whatever they want to do, but the class interface is no more complicated than absolutely necessary.

The desirability of a complete interface seems obvious enough, but why a minimal interface? Why not just give clients everything they ask for, adding functionality until everyone is happy?

Aside from the moral issue — is it really *right* to mollycoddle your clients? — there are definite technical disadvantages to a class interface that is crowded with functions. First, the more functions in an interface, the harder it is for potential clients to understand. The harder it is for them to understand, the more reluctant they will be to learn how to use it. A class with 10 functions looks tractable to most people, but a class with 100 functions is enough to make many programmers run and hide. By expanding the functionality of your class to make it as attractive as possible, you may actually end up discouraging people from learning how to use it.

A large interface can also lead to confusion. Suppose you create a class that supports cognition for an artificial intelligence application. One of your member functions is called `think`, but you later discover that some people want the function to be called `ponder`, and others prefer the name `ruminate`. In an effort to be accommodating, you offer all three functions, even though they do the same thing. Consider then the plight of a potential client of your class who is trying to figure things

out. The client is faced with three different functions, all of which are supposed to do the same thing. Can that really be true? Isn't there some subtle difference between the three, possibly in efficiency or generality or reliability? If not, why are there three different functions? Rather than appreciating your flexibility, such a potential client is likely to wonder what on earth you were thinking (or pondering, or ruminating over).

A second disadvantage to a large class interface is that of maintenance (see Item M32). It's simply more difficult to maintain and enhance a class with many functions than it is a class with few. It is more difficult to avoid duplicated code (with the attendant duplicated bugs), and it is more difficult to maintain consistency across the interface. It's also more difficult to document.

Finally, long class definitions result in long header files. Because header files typically have to be read every time a program is compiled (see Item 34), class definitions that are longer than necessary can incur a substantial penalty in total compile-time over the life of a project.

The long and short of it is that the gratuitous addition of functions to an interface is not without costs, so you need to think carefully about whether the convenience of a new function (a new function can *only* be added for convenience if the interface is already complete) justifies the additional costs in complexity, comprehensibility, maintainability, and compilation speed.

Yet there's no sense in being unduly miserly. It is often justifiable to offer more than a minimal set of functions. If a commonly performed task can be implemented much more efficiently as a member function, that may well justify its addition to the interface. (Then again, it may not. See Item M16.) If the addition of a member function makes the class substantially easier to use, that may be enough to warrant its inclusion in the class. And if adding a member function is likely to prevent client errors, that, too, is a powerful argument for its being part of the interface.

Consider a concrete example: a template for classes that implement arrays with client-defined upper and lower bounds and that offer optional bounds-checking. The beginning of such an array template is shown below:

```
template<class T>
class Array {
public:
  enum BoundsCheckingStatus {NO_CHECK_BOUNDS = 0,
                             CHECK_BOUNDS = 1};


  Array(int lowBound, int highBound,
       BoundsCheckingStatus check = NO_CHECK_BOUNDS);


  Array(const Array& rhs);
```

```
  ~Array();


  Array& operator=(const Array& rhs);


private:
  int lBound, hBound;         // low bound, high bound


  vector<T> data;             // contents of array; see
                              // Item 49 for vector info


  BoundsCheckingStatus checkingBounds;
};
```

The member functions declared so far are the ones that require basically no thinking (or pondering or ruminating). You have a constructor to allow clients to specify each array's bounds, a copy constructor, an assignment operator, and a destructor. In this case, you've declared the destructor nonvirtual, which implies that this class is not to be used as a base class (see Item 14).

The declaration of the assignment operator is actually less clear-cut than it might at first appear. After all, built-in arrays in C++ don't allow assignment, so you might want to disallow it for your `Array` objects, too (see Item 27). On the other hand, the array-like `vector` template (in the standard library — see Item 49) permits assignments between `vector` objects. In this example, you'll follow `vector`'s lead, and that decision, as you'll see below, will affect other portions of the classes's interface.

Old-time C hacks would cringe to see this interface. Where is the support for declaring an array of a particular size? It would be easy enough to add another constructor,

```
Array(int size,
      BoundsCheckingStatus check = NO_CHECK_BOUNDS);
```

but this is not part of a minimal interface, because the constructor taking an upper and lower bound can be used to accomplish the same thing. Nonetheless, it might be a wise political move to humor the old geezers, possibly under the rubric of consistency with the base language.

What other functions do you need? Certainly it is part of a complete interface to index into an

array:

```
// return element for read/write
T& operator[](int index);


// return element for read-only
const T& operator[](int index) const;
```

By declaring the same function twice, once `const` and once non-`const`, you provide support for both `const` and non-`const` `Array` objects. The difference in return types is significant, as is explained in Item 21.

As it now stands, the `Array` template supports construction, destruction, pass-by-value, assignment, and indexing, which may strike you as a complete interface. But look closer. Suppose a client wants to loop through an array of integers, printing out each of its elements, like so:

```
Array<int> a(10, 20);      // bounds on a are 10 to 20


...


for (int i = lower bound of a; i <= upper bound of a; ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

How is the client to get the bounds of `a`? The answer depends on what happens during assignment of `Array` objects, i.e., on what happens inside `Array::operator=`. In particular, if assignment can change the bounds of an `Array` object, you must provide member functions to return the current bounds, because the client has no way of knowing *a priori* what the bounds are at any given point in the program. In the example above, if `a` was the target of an assignment between the time it was defined and the time it was used in the loop, the client would have no way to determine the current bounds of `a`.

On the other hand, if the bounds of an `Array` object cannot be changed during assignment, then the bounds are fixed at the point of definition, and it would be possible (though cumbersome) for a client to keep track of these bounds. In that case, though it would be convenient to offer functions to return the current bounds, such functions would not be part of a truly minimal interface.

Proceeding on the assumption that assignment can modify the bounds of an object, the bounds functions could be declared thus:

```
int lowBound() const;
int highBound() const;
```

Because these functions don't modify the object on which they are invoked, and because you prefer to use `const` whenever you can (see Item 21), these are both declared `const` member functions. Given these functions, the loop above would be written as follows:

```
for (int i = a.lowBound(); i <= a.highBound(); ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

Needless to say, for such a loop to work for an array of objects of type `T`, an `operator<<` function must be defined for objects of type `T`. (That's not quite true. What must exist is an `operator<<` for `T` or for some other type to which `T` may be implicitly converted (see Item M5). But you get the idea.)

Some designers would argue that the `Array` class should also offer a function to return the number of elements in an `Array` object. The number of elements is simply `highBound()-lowBound()+1`, so such a function is not really necessary, but in view of the frequency of off-by-one errors, it might not be a bad idea to add such a function.

Other functions that might prove worthwhile for this class include those for input and output, as well as the various relational operators (e.g., <, >, ==, etc.). None of those functions is part of a minimal interface, however, because they can all be implemented in terms of loops containing calls to `operator[]`.

Speaking of functions like `operator<<`, `operator>>`, and the relational operators, Item 19 discusses why they are frequently implemented as non-member friend functions instead of as member functions. That being the case, don't forget that friend functions are, for all practical purposes, part of a class's interface. That means that friend functions count toward a class interface's completeness and minimalness.

Back to Design and Declaration
Continue to Item 19: Differentiate among member functions, non-member functions, and friend functions.

Back to Item 18: Strive for class interfaces that are complete and minimal.
Continue to Item 20: Avoid data members in the public interface.

Item 19: Differentiate among member functions, non-member functions, and friend functions.

The biggest difference between member functions and non-member functions is that member functions can be virtual and non-member functions can't. As a result, if you have a function that has to be dynamically bound (see Item 38), you've got to use a virtual function, and that virtual function must be a member of some class. It's as simple as that. If your function doesn't need to be virtual, however, the water begins to muddy a bit.

Consider a class for representing rational numbers:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;


private:
  ...
};
```

As it stands now, this is a pretty useless class. (Using the terms of Item 18, the interface is certainly minimal, but it's *far* from complete.) You know you'd like to support arithmetic operations like addition, subtraction, multiplication, etc., but you're unsure whether you should implement them via a member function, a non-member function, or possibly a non-member function that's a friend.

When in doubt, be object-oriented. You know that, say, multiplication of rational numbers is related to the `Rational` class, so try bundling the operation with the class by making it a member function:

```
class Rational {
public:


  ...


  const Rational operator*(const Rational& rhs) const;
};
```

(If you're unsure why this function is declared the way it is — returning a `const` by-value result, but taking a reference-to-`const` as its argument — consult Items 21-23.)

Now you can multiply rational numbers with the greatest of ease:

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);


Rational result = oneHalf * oneEighth;      // fine


result = result * oneEighth;                // fine
```

But you're not satisfied. You'd also like to support mixed-mode operations, where `Rationals` can be multiplied with, for example, `ints`. When you try to do this, however, you find that it works only half the time:

```
result = oneHalf * 2;      // fine


result = 2 * oneHalf;      // error!
```

This is a bad omen. Multiplication is supposed to be commutative, remember?

The source of the problem becomes apparent when you rewrite the last two examples in their equivalent functional form:

```
result = oneHalf.operator*(2);      // fine


result = 2.operator*(oneHalf);      // error!
```

The object `oneHalf` is an instance of a class that contains an `operator*`, so your compilers call that function. However, the integer 2 has no associated class, hence no `operator*` member function. Your compilers will also look for a non-member `operator*` (i.e., one that's in a visible namespace or is global) that can be called like this,

```
result = operator*(2, oneHalf);      // error!
```

but there is no non-member `operator*` taking an `int` and a `Rational`, so the search fails.

Look again at the call that succeeds. You'll see that its second parameter is the integer 2, yet `Rational::operator*` takes a `Rational` object as its argument. What's going on here? Why does 2 work in one position and not in the other?

What's going on is implicit type conversion. Your compilers know you're passing an `int` and the function requires a `Rational`, but they also know that they can conjure up a suitable `Rational` by calling the `Rational` constructor with the `int` you provided, so that's what they do (see Item M19). In other words, they treat the call as if it had been written more or less like this:

```
const Rational temp(2);      // create a temporary
                             // Rational object from 2

result = oneHalf * temp;     // same as
                             // oneHalf.operator*(temp);
```

Of course, they do this only when non-`explicit` constructors are involved, because `explicit` constructors can't be used for implicit conversions; that's what `explicit` means. If `Rational` were defined like this,

```
class Rational {
public:
  explicit Rational(int numerator = 0,     // this ctor is
                    int denominator = 1);  // now explicit
  ...

  const Rational operator*(const Rational& rhs) const;

  ...

};
```

neither of these statements would compile:

```
result = oneHalf * 2;             // error!
result = 2 * oneHalf;             // error!
```

That would hardly qualify as support for mixed-mode arithmetic, but at least the behavior of the two statements would be consistent.

The `Rational` class we've been examining, however, is designed to allow implicit conversions from built-in types to `Rational`s — that's why `Rational`'s constructor isn't declared `explicit`. That being the case, compilers will perform the implicit conversion necessary to allow `result`'s first assignment to compile. In fact, your handy-dandy compilers will perform this kind of implicit type conversion, if it's needed, on *every* parameter of *every* function call. But they will do it only for parameters *listed in the parameter list*, *never* for the object on which a member function is invoked, i.e., the object corresponding to `*this` inside a member function. That's why this call works,

```
result = oneHalf.operator*(2);      // converts int -> Rational
```

and this one does not:

```
result = 2.operator*(oneHalf);      // doesn't convert
                                    // int -> Rational
```

The first case involves a parameter listed in the function declaration, but the second one does not.

Nonetheless, you'd still like to support mixed-mode arithmetic, and the way to do it is by now perhaps clear: make `operator*` a non-member function, thus allowing compilers to perform implicit type conversions on *all* arguments:

```
class Rational {


  ...                               // contains no operator*


};


// declare this globally or within a namespace; see
// Item M20 for why it's written as it is
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
```

```
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}



Rational oneFourth(1, 4);
Rational result;



result = oneFourth * 2;           // fine
result = 2 * oneFourth;           // hooray, it works!
```

This is certainly a happy ending to the tale, but there is a nagging worry. Should `operator*` be made a friend of the `Rational` class?

In this case, the answer is no, because `operator*` can be implemented entirely in terms of the class's public interface. The code above shows one way to do it. Whenever you can avoid friend functions, you should, because, much as in real life, friends are often more trouble than they're worth.

However, it's not uncommon for functions that are not members, yet are still conceptually part of a class interface, to need access to the non-public members of the class.

As an example, let's fall back on a workhorse of this book, the `String` class. If you try to overload `operator>>` and `operator<<` for reading and writing `String` objects, you'll quickly discover that they shouldn't be member functions. If they were, you'd have to put the `String` object on the left when you called the functions:

```
// a class that incorrectly declares operator>> and
// operator<< as member functions
class String {
public:
  String(const char *value);



  ...



  istream& operator>>(istream& input);
  ostream& operator<<(ostream& output);



private:
```

```
  char *data;
};


String s;



s >> cin;                  // legal, but contrary
                           // to convention



s << cout;                 // ditto
```

That would confuse everyone. As a result, these functions shouldn't be member functions. Notice that this is a different case from the one we discussed above. Here the goal is a natural calling syntax; earlier we were concerned about implicit type conversions.

If you were designing these functions, you'd come up with something like this:

```
istream& operator>>(istream& input, String& string)
{
  delete [] string.data;



  read from input into some memory, and make string.data
  point to it



  return input;
}



ostream& operator<<(ostream& output,
                    const String& string)
{
  return output << string.data;
}
```

Notice that both functions need access to the `data` field of the `String` class, a field that's private. However, you already know that you have to make these functions non-members. You're boxed into a corner and have no choice: non-member functions with a need for access to non-public members of a class must be made friends of that class.

The lessons of this Item are summarized below, in which it is assumed that `f` is the function you're trying to declare properly and `C` is the class to which it is conceptually related:

- **Virtual functions must be members.** If `f` needs to be virtual, make it a member function of `C`.
- **`operator>>` and `operator<<` are never members.** If `f` is `operator>>` or `operator<<`, make `f` a non-member function. If, in addition, `f` needs access to non-public members of `C`, make `f` a friend of `C`.
- **Only non-member functions get type conversions on their left-most argument.** If `f` needs type conversions on its left-most argument, make `f` a non-member function. If, in addition, `f` needs access to non-public members of `C`, make `f` a friend of `C`.
- **Everything else should be a member function.** If none of the other cases apply, make `f` a member function of `C`.

Back to Item 18: Strive for class interfaces that are complete and minimal.
Continue to Item 20: Avoid data members in the public interface.

Back to Item 19: Differentiate among member functions, non-member functions, and friend functions.
Continue to Item 21: Use const whenever possible.

Item 20: Avoid data members in the public interface.

First, let's look at this issue from the point of view of consistency. If everything in the public interface is a function, clients of your class won't have to scratch their heads trying to remember whether to use parentheses when they want to access a member of your class. They'll just *do* it, because everything is a function. Over the course of a lifetime, that can save a lot of head scratching.

You don't buy the consistency argument? How about the fact that using functions gives you much more precise control over the accessibility of data members? If you make a data member public, everybody has read/write access to it, but if you use functions to get and set its value, you can implement no access, read-only access, and read-write access. Heck, you can even implement write-only access if you want to:

```
class AccessLevels {
public:
  int getReadOnly() const{ return readOnly; }


  void setReadWrite(int value) { readWrite = value; }
  int getReadWrite() const { return readWrite; }


  void setWriteOnly(int value) { writeOnly = value; }


private:
  int noAccess;                  // no access to this   int


  int readOnly;                  // read-only access to
                                 // this int


  int readWrite;                 // read-write access to
                                 // this int


  int writeOnly;                 // write-only access to
```

```
                                        // this int
};
```

Still not convinced? Then it's time to bring out the big gun: functional abstraction. If you implement access to a data member through a function, you can later replace the data member with a computation, and nobody using your class will be any the wiser.

For example, suppose you are writing an application in which some automated equipment is monitoring the speed of passing cars. As each car passes, its speed is computed, and the value is added to a collection of all the speed data collected so far:

```
class SpeedDataCollection {
public:
  void addValue(int speed);       // add a new data value


  double averageSoFar() const;    // return average speed
};
```

Now consider the implementation of the member function `averageSoFar` (see also Item M18). One way to implement it is to have a data member in the class that is a running average of all the speed data so far collected. Whenever `averageSoFar` is called, it just returns the value of that data member. A different approach is to have `averageSoFar` compute its value anew each time it's called, something it could do by examining each data value in the collection. (For a more general discussion of these two approaches, see Items M17 and M18.)

The first approach — keeping a running average — makes each `SpeedDataCollection` object bigger, because you have to allocate space for the data member holding the running average. However, `averageSoFar` can be implemented very efficiently; it's just an inline function (see Item 33) that returns the value of the data member. Conversely, computing the average whenever it's requested will make `averageSoFar` run slower, but each `SpeedDataCollection` object will be smaller.

Who's to say which is best? On a machine where memory is tight, and in an application where averages are needed only infrequently, computing the average each time is a better solution. In an application where averages are needed frequently, speed is of the essence, and memory is not an issue, keeping a running average is preferable. The important point is that by accessing the average through a member function, you can use *either* implementation, a valuable source of flexibility that you wouldn't have if you made a decision to include the running average data member in the public interface.

The upshot of all this is that you're just asking for trouble by putting data members in the public interface, so play it safe by hiding all your data members behind a wall of functional abstraction. If you do it *now*, we'll throw in consistency and fine-grained access control at no extra cost!

Back to Item 19: Differentiate among member functions, non-member functions, and friend functions.
Continue to Item 21: Use const whenever possible.

Back to Item 20: Avoid data members in the public interface.
Continue to Item 22: Prefer pass-by-reference to pass-by-value.

Item 21: Use `const` whenever possible.

The wonderful thing about `const` is that it allows you to specify a certain semantic constraint — a particular object should *not* be modified — and compilers will enforce that constraint. It allows you to communicate to both compilers and other programmers that a value should remain invariant. Whenever that is true, you should be sure to say so explicitly, because that way you enlist your compilers' aid in making sure the constraint isn't violated.

The `const` keyword is remarkably versatile. Outside of classes, you can use it for global or namespace constants (see Items 1 and 47) and for static objects (local to a file or a block). Inside classes, you can use it for both static and nonstatic data members (see also Item 12).

For pointers, you can specify whether the pointer itself is `const`, the data it points to is `const`, both, or neither:

```
char *p             = "Hello";          // non-const pointer,
                                        // non-const data[5]


const char *p       = "Hello";          // non-const pointer,
                                        // const data


char * const p      = "Hello";          // const pointer,
                                        // non-const data


const char * const p = "Hello";         // const pointer,
                                        // const data
```

This syntax isn't quite as capricious as it looks. Basically, you mentally draw a vertical line through the asterisk of a pointer declaration, and if the word `const` appears to the left of the line, what's *pointed to* is constant; if the word `const` appears to the right of the line, the *pointer itself* is constant; if `const` appears on both sides of the line, both are constant.

When what's pointed to is constant, some programmers list `const` before the type name. Others list

it after the type name but before the asterisk. As a result, the following functions take the same parameter type:

```
class Widget { ... };


void f1(const Widget *pw);      // f1 takes a pointer to a
                                // constant Widget object


void f2(Widget const *pw);      // so does f2
```

Because both forms exist in real code, you should accustom yourself to both of them.

Some of the most powerful uses of `const` stem from its application to function declarations. Within a function declaration, `const` can refer to the function's return value, to individual parameters, and, for member functions, to the function as a whole.

Having a function return a constant value often makes it possible to reduce the incidence of client errors without giving up safety or efficiency. In fact, as Item 29 demonstrates, using `const` with a return value can make it possible to *improve* the safety and efficiency of a function that would otherwise be problematic.

For example, consider the declaration of the `operator*` function for rational numbers that is introduced in Item 19:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Many programmers squint when they first see this. Why should the result of `operator*` be a `const` object? Because if it weren't, clients would be able to commit atrocities like this:

```
Rational a, b, c;


...
```

```
(a * b) = c;      // assign to the product
                  // of a*b!
```

I don't know why any programmer would want to make an assignment to the product of two numbers, but I do know this: it would be flat-out illegal if `a`, `b`, and `c` were of a built-in type. One of the hallmarks of good user-defined types is that they avoid gratuitous behavioral incompatibilities with the built-ins, and allowing assignments to the product of two numbers seems pretty gratuitous to me. Declaring `operator*`'s return value `const` prevents it, and that's why It's The Right Thing To Do.

There's nothing particularly new about `const` parameters — they act just like local `const` objects. (See Item M19, however, for a discussion of how `const` parameters can lead to the creation of temporary objects.) Member functions that are `const`, however, are a different story.

The purpose of `const` member functions, of course, is to specify which member functions may be invoked on `const` objects. Many people overlook the fact that member functions differing *only* in their constness can be overloaded, however, and this is an important feature of C++. Consider the `String` class once again:

```
class String {
public:


  ...


  // operator[] for non-const objects
  char& operator[](int position)
  { return data[position]; }


  // operator[] for const objects
  const char& operator[](int position) const
  { return data[position]; }


private:
  char *data;
};


String s1 = "Hello";
cout << s1[0];                  // calls non-const
                                // String::operator[]
const String s2 = "World";
```

```
cout << s2[0];                  // calls const
                                // String::operator[]
```

By overloading `operator[]` and giving the different versions different return values, you are able to have `const` and non-`const` `String`s handled differently:

```
String s = "Hello";                     // non-const String object

cout << s[0];                           // fine – reading a
                                        // non-const String

s[0] = 'x';                             // fine – writing a
                                        // non-const String

const String cs = "World";              // const String object

cout << cs[0];                          // fine – reading a
                                        // const String

cs[0] = 'x';                            // error! – writing a
                                        // const String
```

By the way, note that the error here has only to do with the *return value* of the `operator[]` that is called; the calls to `operator[]` themselves are all fine. The error arises out of an attempt to make an assignment to a `const char&`, because that's the return value from the `const` version of `operator[]`.

Also note that the return type of the non-`const` `operator[]` must be a *reference* to a `char` — a `char` itself will not do. If `operator[]` did return a simple `char`, statements like this wouldn't compile:

```
s[0] = 'x';
```

That's because it's never legal to modify the return value of a function that returns a built-in type. Even if it were legal, the fact that C++ returns objects by value (see Item 22) would mean that a *copy* of `s.data[0]` would be modified, not `s.data[0]` itself, and that's not the behavior you want, anyway.

Let's take a brief time-out for philosophy. What exactly does it mean for a member function to be `const`? There are two prevailing notions: bitwise constness and conceptual constness.

The bitwise `const` camp believes that a member function is `const` if and only if it doesn't modify any of the object's data members (excluding those that are static), i.e., if it doesn't modify any of the bits inside the object. The nice thing about bitwise constness is that it's easy to detect violations: compilers just look for assignments to data members. In fact, bitwise constness is C++'s definition of constness, and a `const` member function isn't allowed to modify any of the data members of the object on which it is invoked.

Unfortunately, many member functions that don't act very `const` pass the bitwise test. In particular, a member function that modifies what a pointer *points to* frequently doesn't act `const`. But if only the *pointer* is in the object, the function is bitwise `const`, and compilers won't complain. That can lead to counterintuitive behavior:

```
class String {
public:
  // the constructor makes data point to a copy
  // of what value points to
  String(const char *value);


  ...


  operator char *() const { return data;}


private:
  char *data;
};



const String s = "Hello";      // declare constant object



char *nasty = s;               // calls op char*() const
```

```
*nasty = 'M';                  // modifies s.data[0]



cout << s;                     // writes "Mello"
```

Surely there is something wrong when you create a constant object with a particular value and you invoke only `const` member functions on it, yet you are still able to change its value! (For a more detailed discussion of this example, see Item 29.)

This leads to the notion of conceptual constness. Adherents to this philosophy argue that a `const` member function might modify some of the bits in the object on which it's invoked, but only in ways that are undetectable by clients. For example, your `String` class might want to cache the length of the object whenever it's requested (see Item M18):

```
class String {
public:
  // the constructor makes data point to a copy
  // of what value points to
  String(const char *value): lengthIsValid(false) { ... }



  ...


  size_t length() const;



private:
  char *data;



  size_t dataLength;               // last calculated length
                                   // of string



  bool lengthIsValid;              // whether length is
                                   // currently valid
};



size_t String::length() const
```

```
{
  if (!lengthIsValid) {
    dataLength = strlen(data);      // error!
    lengthIsValid = true;           // error!
  }


  return dataLength;
}
```

This implementation of `length` is certainly not bitwise const — both `dataLength` and `lengthIsValid` may be modified — yet it seems as though it should be valid for `const String` objects. Compilers, you will find, respectfully disagree; they insist on bitwise constness. What to do?

The solution is simple: take advantage of the `const`-related wiggle room the C++ standardization committee thoughtfully provided for just these types of situations. That wiggle room takes the form of the keyword `mutable`. When applied to nonstatic data members, `mutable` frees those members from the constraints of bitwise constness:

```
class String {
public:


  ...    // same as above


private:
  char *data;



  mutable size_t dataLength;            // these data members are
                                        // now mutable; they may be
  mutable bool lengthIsValid;           // modified anywhere, even
                                        // inside const member
};                                      // functions


size_t String::length() const
{
  if (!lengthIsValid) {
    dataLength = strlen(data);    // now fine
    lengthIsValid = true;         // also fine
  }
```

```
  return dataLength;
}
```

`mutable` is a wonderful solution to the bitwise-constness-is-not-quite-what-I-had-in-mind problem, but it was added to C++ relatively late in the standardization process, so your compilers may not support it yet. If that's the case, you must descend into the dark recesses of C++, where life is cheap and constness may be cast away.

Inside a member function of class `C`, the `this` pointer behaves as if it had been declared as follows:

```
C * const this;                         // for non-const member
                                        // functions


const C * const this;                   // for const member
                                        // functions
```

That being the case, all you have to do to make the problematic version of `String::length` (i.e., the one you could fix with `mutable` if your compilers supported it) valid for both `const` and non-`const` objects is to change the type of `this` from `const C * const` to `C * const`. You can't do that directly, but you can fake it by initializing a local pointer to point to the same object as `this` does. Then you can access the members you want to modify through the local pointer:

```
size_t String::length() const
{
  // make a local version of this that's
  // not a pointer-to-const
  String * const localThis =
    const_cast<String * const>(this);


  if (!lengthIsValid) {
    localThis->dataLength = strlen(data);
    localThis->lengthIsValid = true;
  }


  return dataLength;
}
```

Pretty this ain't, but sometimes a programmer's just gotta do what a programmer's gotta do.

Unless, of course, it's not guaranteed to work, and sometimes the old cast-away-constness trick isn't. In particular, if the object `this` points to is truly `const`, i.e., was declared `const` at its point of definition, the results of casting away its constness are undefined. If you want to cast away constness in one of your member functions, you'd best be sure that the object you're doing the casting on wasn't originally defined to be `const`.

There is one other time when casting away constness may be both useful and safe. That's when you have a `const` object you want to pass to a function taking a non-`const` parameter, and *you know the parameter won't be modified inside the function*. The second condition is important, because it is always safe to cast away the constness of an object that will only be read — not written — even if that object was originally defined to be `const`.

For example, some libraries have been known to incorrectly declare the `strlen` function as follows:

```
size_t strlen(char *s);
```

Certainly `strlen` isn't going to modify what `s` points to — at least not the `strlen` I grew up with. Because of this declaration, however, it would be invalid to call it on pointers of type `const char *`. To get around the problem, you can safely cast away the constness of such pointers when you pass them to `strlen`:

```
const char *klingonGreeting = "nuqneH";        // "nuqneH" is
                                               // "Hello" in
                                               // Klingon
size_t length =
  strlen(const_cast<char*>(klingonGreeting));
```

Don't get cavalier about this, though. It is guaranteed to work only if the function being called, `strlen` in this case, doesn't try to modify what its parameter points to.

Back to Item 20: Avoid data members in the public interface.
Continue to Item 22: Prefer pass-by-reference to pass-by-value.

---

[5] According to the ◦C++ standard, the type of `"Hello"` is `const char []`, a type that's almost always treated as `const char*`. We'd therefore expect it to be a violation of `const` correctness to initialize a `char*` variable with a string literal like `"Hello"`. The practice is so common in C, however, that the standard grants a special dispensation for initializations like this. Nevertheless, you should try to avoid them, because they're deprecated.
Return

Back to Item 21: Use const whenever possible.
Continue to Item 23: Don't try to return a reference when you must return an object.

Item 22: Prefer pass-by-reference to pass-by-value.

In C, everything is passed by value, and C++ honors this heritage by adopting the pass-by-value convention as its default. Unless you specify otherwise, function parameters are initialized with *copies* of the actual arguments, and function callers get back a *copy* of the value returned by the function.

As I pointed out in the Introduction to this book, the meaning of passing an object by value is defined by the copy constructor of that object's class. This can make pass-by-value an extremely expensive operation. For example, consider the following (rather contrived) class hierarchy:

```
class Person {
public:
  Person();                          // parameters omitted for
                                     // simplicity
  ~Person();


  ...


private:
  string name, address;
};


class Student: public Person {
public:
  Student();                         // parameters omitted for
                                     // simplicity
  ~Student();


  ...


private:
  string schoolName, schoolAddress;
};
```

Now consider a simple function `returnStudent` that takes a `Student` argument (by value) and immediately returns it (also by value), plus a call to that function:

```
Student returnStudent(Student s) { return s; }


Student plato;                      // Plato studied under
                                    // Socrates


returnStudent(plato);               // call returnStudent
```

What happens during the course of this innocuous-looking function call?

The simple explanation is this: the `Student` copy constructor is called to initialize `s` with `plato`. Then the `Student` copy constructor is called again to initialize the object returned by the function with `s`. Next, the destructor is called for `s`. Finally, the destructor is called for the object returned by `returnStudent`. So the cost of this do-nothing function is two calls to the `Student` copy constructor and two calls to the `Student` destructor.

But wait, there's more! A `Student` object has two `string` objects within it, so every time you construct a `Student` object you must also construct two `string` objects. A `Student` object also inherits from a `Person` object, so every time you construct a `Student` object you must also construct a `Person` object. A `Person` object has two additional `string` objects inside it, so each `Person` construction also entails two more `string` constructions. The end result is that passing a `Student` object by value leads to one call to the `Student` copy constructor, one call to the `Person` copy constructor, and four calls to the `string` copy constructor. When the copy of the `Student` object is destroyed, each constructor call is matched by a destructor call, so the overall cost of passing a `Student` by value is six constructors and six destructors. Because the function `returnStudent` uses pass-by-value twice (once for the parameter, once for the return value), the complete cost of a call to that function is *twelve* constructors and *twelve* destructors!

In fairness to the C++ compiler-writers of the world, this is a worst-case scenario. Compilers are allowed to eliminate some of these calls to copy constructors. (The ◦C++ standard — see Item 50 — describes the precise conditions under which they are allowed to perform this kind of magic, and Item M20 gives examples). Some compilers take advantage of this license to optimize. Until such optimizations become ubiquitous, however, you've got to be wary of the cost of passing objects by value.

To avoid this potentially exorbitant cost, you need to pass things not by value, but by reference:

```
const Student& returnStudent(const Student& s)
```

```
{ return s; }
```

This is much more efficient: no constructors or destructors are called, because no new objects are being created.

Passing parameters by reference has another advantage: it avoids what is sometimes called the "slicing problem." When a derived class object is passed as a base class object, all the specialized features that make it behave like a derived class object are "sliced" off, and you're left with a simple base class object. This is almost never what you want. For example, suppose you're working on a set of classes for implementing a graphical window system:

```
class Window {
public:
  string name() const;             // return name of window
  virtual void display() const;    // draw window and contents
};


class WindowWithScrollBars: public Window {
public:
  virtual void display() const;
};
```

All `Window` objects have a name, which you can get at through the `name` function, and all windows can be displayed, which you can bring about by invoking the `display` function. The fact that `display` is virtual tells you that the way in which simple base class `Window` objects are displayed is apt to differ from the way in which the fancy, high-priced `WindowWithScrollBars` objects are displayed (see Items 36, 37, and M33).

Now suppose you'd like to write a function to print out a window's name and then display the window. Here's the *wrong* way to write such a function:

```
// a function that suffers from the slicing problem
void printNameAndDisplay(Window w)
{
  cout << w.name();
  w.display();
}
```

Consider what happens when you call this function with a `WindowWithScrollBars` object:

```
WindowWithScrollBars wwsb;
```

```
printNameAndDisplay(wwsb);
```

The parameter `w` will be constructed — it's passed by value, remember? — as a *`Window`* object, and all the specialized information that made `wwsb` act like a `WindowWithScrollBars` object will be sliced off. Inside `printNameAndDisplay`, `w` will always act like an object of class `Window` (because it *is* an object of class `Window`), regardless of the type of object that is passed to the function. In particular, the call to `display` inside `printNameAndDisplay` will *always* call `Window::display`, never `WindowWithScrollBars::display`.

The way around the slicing problem is to pass `w` by reference:

```
// a function that doesn't suffer from the slicing problem
void printNameAndDisplay(const Window& w)
{
  cout << w.name();
  w.display();
}
```

Now `w` will act like whatever kind of window is actually passed in. To emphasize that `w` isn't modified by this function even though it's passed by reference, you've followed the advice of Item 21 and carefully declared it to be `const`; how good of you.

Passing by reference is a wonderful thing, but it leads to certain complications of its own, the most notorious of which is aliasing, a topic that is discussed in Item 17. In addition, it's important to recognize that you sometimes *can't* pass things by reference; see Item 23. Finally, the brutal fact of the matter is that references are almost always *implemented* as pointers, so passing something by reference usually means really passing a pointer. As a result, if you have a small object — an `int`, for example — it may actually be more efficient to pass it by value than to pass it by reference.

Back to Item 21: Use const whenever possible.
Continue to Item 23: Don't try to return a reference when you must return an object.

Back to Item 22: Prefer pass-by-reference to pass-by-value.
Continue to Item 24: Choose carefully between function overloading and parameter defaulting.

Item 23: Don't try to return a reference when you must return an object.

It is said that Albert Einstein once offered this advice: make things as simple as possible, but no simpler. The C++ analogue might well be to make things as efficient as possible, but no more efficient.

Once programmers grasp the efficiency implications of pass-by-value for objects (see Item 22), they become crusaders, determined to root out the evil of pass-by-value wherever it may hide. Unrelenting in their pursuit of pass-by-reference purity, they invariably make a fatal mistake: they start to pass references to objects that don't exist. This is not a good thing.

Consider a class for representing rational numbers, including a friend function (see Item 19) for multiplying two rationals together:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);


  ...


private:
  int n, d;              // numerator and denominator


friend
  const Rational                      // see Item 21 for why
    operator*(const Rational& lhs,    // the return value is
              const Rational& rhs)    // const
};


inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Clearly, this version of `operator*` is returning its result object by value, and you'd be shirking your

professional duties if you failed to worry about the cost of that object's construction and destruction. Another thing that's clear is that you're cheap and you don't want to pay for such a temporary object (see Item M19) if you don't have to. So the question is this: do you have to pay?

Well, you don't have to if you can return a reference instead. But remember that a reference is just a *name*, a name for some *existing* object. Whenever you see the declaration for a reference, you should immediately ask yourself what it is another name for, because it must be another name for *something* (see Item M1). In the case of `operator*`, if the function is to return a reference, it must return a reference to some other `Rational` object that already exists and that contains the product of the two objects that are to be multiplied together.

There is certainly no reason to expect that such an object exists prior to the call to `operator*`. That is, if you have

```
Rational a(1, 2);               // a = 1/2
Rational b(3, 5);               // b = 3/5
Rational c = a * b;             // c should be 3/10
```

it seems unreasonable to expect that there already exists a rational number with the value three-tenths. No, if `operator*` is to return a reference to such a number, it must create that number object itself.

A function can create a new object in only two ways: on the stack or on the heap. Creation on the stack is accomplished by defining a local variable. Using that strategy, you might try to write your `operator*` as follows:

```
// the first wrong way to write this function
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

You can reject this approach out of hand, because your goal was to avoid a constructor call, and `result` will have to be constructed just like any other object. In addition, this function has a more serious problem in that it returns a reference to a local object, an error that is discussed in depth in Item 31.

That leaves you with the possibility of constructing an object on the heap and then returning a reference to it. Heap-based objects come into being through the use of `new`. This is how you might write `operator*` in that case:

```
// the second wrong way to write this function
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  Rational *result =
    new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

Well, you *still* have to pay for a constructor call, because the memory allocated by `new` is initialized by calling an appropriate constructor (see Items 5 and M8), but now you have a different problem: who will apply `delete` to the object that was conjured up by your use of `new`?

In fact, this is a guaranteed memory leak. Even if callers of `operator*` could be persuaded to take the address of the function's result and use `delete` on it (astronomically unlikely — Item 31 shows what the code would have to look like), complicated expressions would yield unnamed temporaries that programmers would never be able to get at. For example, in

```
Rational w, x, y, z;


w = x * y * z;
```

both calls to `operator*` yield unnamed temporaries that the programmer never sees, hence can never delete. (Again, see Item 31.)

But perhaps you think you're smarter than the average bear — or the average programmer. Perhaps you notice that both the on-the-stack and the on-the-heap approaches suffer from having to call a constructor for each result returned from `operator*`. Perhaps you recall that our initial goal was to avoid such constructor invocations. Perhaps you think you know of a way to avoid all but one constructor call. Perhaps the following implementation occurs to you, an implementation based on `operator*` returning a reference to a *static* `Rational` object, one defined *inside* the function:

```
// the third wrong way to write this function
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  static Rational result;      // static object to which a
                               // reference will be returned


  somehow multiply lhs and rhs and put the
  resulting value inside result;
```

```
  return result;
}
```

This looks promising, though when you try to compose real C++ for the italicized pseudocode above, you'll find that it's all but impossible to give `result` the correct value without invoking a `Rational` constructor, and avoiding such a call is the whole reason for this game. Let us posit that you manage to find a way, however, because no amount of cleverness can ultimately save this star-crossed design.

To see why, consider this perfectly reasonable client code:

```
bool operator==(const Rational& lhs,      // an operator==
                const Rational& rhs);     // for Rationals


Rational a, b, c, d;


...


if ((a * b) == (c * d)) {


  do whatever's appropriate when the products are equal;


} else {


  do whatever's appropriate when they're not;


}
```

Now ponder this: the expression `((a*b) == (c*d))` will *always* evaluate to `true`, regardless of the values of `a`, `b`, `c`, and `d`!

It's easiest to understand this vexing behavior by rewriting the test for equality in its equivalent

functional form:

```
if (operator==(operator*(a, b), operator*(c, d)))
```

Notice that when `operator==` is called, there will already be *two* active calls to `operator*`, each of which will return a reference to the static `Rational` object inside `operator*`. Thus, `operator==` will be asked to compare the value of the static `Rational` object inside `operator*` with the value of the static `Rational` object inside `operator*`. It would be surprising indeed if they did not compare equal. Always.

With luck, this is enough to convince you that returning a reference from a function like `operator*` is a waste of time, but I'm not so naive as to believe that luck is always sufficient. Some of you — and you know who you are — are at this very moment thinking, "Well, if *one* static isn't enough, maybe a static *array* will do the trick..."

Stop. Please. Haven't we suffered enough already?

I can't bring myself to dignify this design with example code, but I can sketch why even *entertaining* the notion should cause you to blush in shame. First, you must choose *n*, the size of the array. If *n* is too small, you may run out of places to store function return values, in which case you'll have gained nothing over the single-`static` design we just discredited. But if *n* is too big, you'll decrease the performance of your program, because *every* object in the array will be constructed the first time the function is called. That will cost you *n* constructors and *n* destructors, even if the function in question is called only once. If "optimization" is the process of improving software performance, this kind of thing should be called "pessimization." Finally, think about how you'd put the values you need into the array's objects and what it would cost you to do it. The most direct way to move a value between objects is via assignment, but what is the cost of an assignment? In general, it's about the same as a call to a destructor (to destroy the old value) plus a call to a constructor (to copy over the new value). But your goal is to avoid the costs of construction and destruction! Face it: this approach just isn't going to pan out.

No, the right way to write a function that must return a new object is to have that function return a new object. For `Rational`'s `operator*`, that means either the following code (which we first saw back on page 102) or something essentially equivalent:

```
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Sure, you may incur the cost of constructing and destructing `operator*`'s return value, but in the long run, that's a small price to pay for correct behavior. Besides, the bill that so terrifies you may never arrive. Like all programming languages, C++ allows compiler implementers to apply certain optimizations to improve the performance of the generated code, and it turns out that in some cases, `operator*`'s return value can be safely eliminated (see Item M20). When compilers take advantage of that fact (and current compilers often do), your program continues to behave the way it's supposed to, it just does it faster than you expected.

It all boils down to this: when deciding between returning a reference and returning an object, your job is to make the choice that does the right thing. Let your compiler vendors wrestle with figuring out how to make that choice as inexpensive as possible.

Back to Item 22: Prefer pass-by-reference to pass-by-value.
Continue to Item 24: Choose carefully between function overloading and parameter defaulting.

Back to Item 23: Don't try to return a reference when you must return an object.
Continue to Item 25: Avoid overloading on a pointer and a numerical type.

Item 24: Choose carefully between function overloading and parameter defaulting.

The confusion over function overloading and parameter defaulting stems from the fact that they both allow a single function name to be called in more than one way:

```
void f();                               // f is overloaded
void f(int x);



f();                                    // calls f()
f(10);                                  // calls f(int)



void g(int x = 0);                      // g has a default
                                        // parameter value



g();                                    // calls g(0)
g(10);                                  // calls g(10)
```

So which should be used when?

The answer depends on two other questions. First, is there a value you can use for a default? Second, how many algorithms do you want to use? In general, if you can choose a reasonable default value and you want to employ only a single algorithm, you'll use default parameters (see also Item 38). Otherwise you'll use function overloading.

Here's a function to compute the maximum of up to five `int`s. This function uses — take a deep breath and steel yourself — `std::numeric_limits<int>::min()` as a default parameter value. I'll have more to say about that in a moment, but first, here's the code:

```
int max(int a,
        int b = std::numeric_limits<int>::min(),
        int c = std::numeric_limits<int>::min(),
        int d = std::numeric_limits<int>::min(),
```

```
        int e = std::numeric_limits<int>::min())
{
  int temp = a > b ? a : b;
  temp = temp > c ? temp : c;
  temp = temp > d ? temp : d;
  return temp > e ? temp : e;
}
```

Now, calm yourself. `std::numeric_limits<int>::min()` is just the fancy new-fangled way the standard C++ library says what C says via the `INT_MIN` macro in `<limits.h>`: it's the minimum possible value for an `int` in whatever compiler happens to be processing your C++ source code. True, it's a deviation from the terseness for which C is renowned, but there's a method behind all those colons and other syntactic strychnine.

Suppose you'd like to write a function template taking any built-in numeric type as its parameter, and you'd like the functions generated from the template to print the minimum value representable by their instantiation type. Your template would look something like this:

```
template<class T>
void printMinimumValue()
{
  cout << the minimum value representable by T;
}
```

This is a difficult function to write if all you have to work with is `<limits.h>` and `<float.h>`. You don't know what `T` is, so you don't know whether to print out `INT_MIN` or `DBL_MIN` or what.

To sidestep these difficulties, the standard C++ library (see Item 49) defines in the header `<limits>` a class template, `numeric_limits`, which itself defines several static member functions. Each function returns information about the type instantiating the template. That is, the functions in `numeric_limits<int>` return information about type `int`, the functions in `numeric_limits<double>` return information about type `double`, etc. Among the functions in `numeric_limits` is `min`. `min` returns the minimum representable value for the instantiating type, so `numeric_limits<int>::min()` returns the minimum representable integer value.

Given `numeric_limits` (which, like nearly everything in the standard library, is in namespace `std` — see Item 28; `numeric_limits` itself is in the header `<limits>`), writing `printMinimumValue` is as easy as can be:

```
template<class T>
void printMinimumValue()
{
  cout << std::numeric_limits<T>::min();
}
```

This `numeric_limits`-based approach to specifying type-dependent constants may look expensive, but it's not. That's because the long-windedness of the source code fails to be reflected in the resultant object code. In fact, calls to functions in `numeric_limits` generate no instructions at all. To see how that can be, consider the following, which is an obvious way to implement `numeric_limits<int>::min`:

```
#include <limits.h>


namespace std {


  inline int numeric_limits<int>::min() throw ()
  { return INT_MIN; }


}
```

Because this function is declared inline, calls to it should be replaced by its body (see Item 33). That's just `INT_MIN`, which is itself a simple `#define` for some implementation-defined constant. So even though the `max` function at the beginning of this Item looks like it's making a function call for each default parameter value, it's just using a clever way of referring to a type-dependent constant, in this case the value of `INT_MIN`. Such efficient cleverness abounds in C++'s standard library. You really should read Item 49.

Getting back to the `max` function, the crucial observation is that `max` uses the same (rather inefficient) algorithm to compute its result, regardless of the number of arguments provided by the caller. Nowhere in the function do you attempt to figure out which parameters are "real" and which are defaults. Instead, you have chosen a default value that cannot possibly affect the validity of the computation for the algorithm you're using. That's what makes the use of default parameter values a viable solution.

For many functions, there is no reasonable default value. For example, suppose you want to write a function to compute the average of up to five `int`s. You can't use default parameter values here, because the result of the function is dependent on the number of parameters passed in: if 3 values are passed in, you'll divide their sum by 3; if 5 values are passed in, you'll divide their sum by 5. Furthermore, there is no "magic number" you can use as a default to indicate that a parameter wasn't actually provided by the client, because all possible `int`s are valid values for the parameters. In this case, you have no choice: you *must* use overloaded functions:

```
double avg(int a);
double avg(int a, int b);
double avg(int a, int b, int c);
double avg(int a, int b, int c, int d);
double avg(int a, int b, int c, int d, int e);
```

The other case in which you need to use overloaded functions occurs when you want to accomplish a particular task, but the algorithm that you use depends on the inputs that are given. This is commonly the case with constructors: a default constructor will construct an object from scratch, whereas a copy constructor will construct one from an existing object:

```
// A class for representing natural numbers
class Natural {
public:
  Natural(int initValue);
  Natural(const Natural& rhs);



private:
  unsigned int value;



  void init(int initValue);
  void error(const string& msg);
};



inline
void Natural::init(int initValue) { value = initValue; }



Natural::Natural(int initValue)
{
  if (initValue > 0) init(initValue);
  else error("Illegal initial value");
}



inline Natural::Natural(const Natural& x)
{ init(x.value); }
```

The constructor taking an `int` has to perform error checking, but the copy constructor doesn't, so two different functions are needed. That means overloading. However, note that both functions must assign an initial value for the new object. This could lead to code duplication in the two constructors, so you maneuver around that problem by writing a private member function `init` that contains the code common to the two constructors. This tactic — using overloaded functions that

call a common underlying function for some of their work — is worth remembering, because it's frequently useful (see e.g., Item 12).

Back to Item 23: Don't try to return a reference when you must return an object.
Continue to Item 25: Avoid overloading on a pointer and a numerical type.

Back to Item 24: Choose carefully between function overloading and parameter defaulting.
Continue to Item 26: Guard against potential ambiguity.

Item 25: Avoid overloading on a pointer and a numerical type.

Trivia question for the day: what is zero?

More specifically, what will happen here?

```
void f(int x);
void f(string *ps);


f(0);                       // calls f(int) or f(string*)?
```

The answer is that `0` is an `int` — a literal integer constant, to be precise — so `f(int)` will always be called. Therein lies the problem, because that's not what people always want. This is a situation unique in the world of C++: a place where people think a call should be ambiguous, but compilers do not.

It would be nice if you could somehow tiptoe around this problem by use of a symbolic name, say, `NULL` for null pointers, but that turns out to be a lot tougher than you might imagine.

Your first inclination might be to declare a constant called `NULL`, but constants have types, and what type should `NULL` have? It needs to be compatible with all pointer types, but the only type satisfying that requirement is `void*`, and you can't pass `void*` pointers to typed pointers without an explicit cast. Not only is that ugly, at first glance it's not a whole lot better than the original situation:

```
void * const NULL = 0;            // potential NULL definition


f(0);                             // still calls f(int)
f(static_cast<string*>(NULL));    // calls f(string*)
f(static_cast<string*>(0));       // calls f(string*)
```

On second thought, however, the use of `NULL` as a `void*` constant is a shade better than what you started with, because you avoid ambiguity if you use only `NULL` to indicate null pointers:

```
f(0);                              // calls f(int)
f(NULL);                           // error! — type mis-match
f(static_cast<string*>(NULL));     // okay, calls f(string*)
```

At least now you've traded a runtime error (the call to the "wrong" `f` for 0) for a compile-time error (the attempt to pass a `void*` into a `string*` parameter). This improves matters somewhat (see Item 46), but the cast is still unsatisfying.

If you shamefacedly crawl back to the preprocessor, you find that it doesn't really offer a way out, either, because the obvious choices seem to be

```
#define NULL 0
```

and

```
#define NULL ((void*) 0)
```

and the first possibility is just the literal `0`, which is fundamentally an integer constant (your original problem, as you'll recall), while the second possibility gets you back into the trouble with passing `void*` pointers to typed pointers.

If you've boned up on the rules governing type conversions, you may know that C++ views a conversion from a `long int` to an `int` as neither better nor worse than a conversion from the `long int` `0` to the null pointer. You can take advantage of that to introduce the ambiguity into the `int`/pointer question you probably believe should be there in the first place:

```
#define NULL 0L            // NULL is now a long int


void f(int x);
void f(string *p);


f(NULL);                   // error! — ambiguous
```

However, this fails to help if you overload on a `long int` and a pointer:

```
#define NULL 0L
```

```
void f(long int x);        // this f now takes a long
void f(string *p);


f(NULL);                   // fine, calls f(long int)
```

In practice, this is probably safer than defining `NULL` to be an `int`, but it's more a way of moving the problem around than of eliminating it.

The problem can be exterminated, but it requires the use of a late-breaking addition to the language: *member function templates* (often simply called *member templates*). Member function templates are exactly what they sound like: templates within classes that generate member functions for those classes. In the case of `NULL`, you want an object that acts like the expression `static_cast<T*>(0)` for every type `T`. That suggests that `NULL` should be an object of a class containing an implicit conversion operator for every possible pointer type. That's a lot of conversion operators, but a member template lets you force C++ into generating them for you:

```
// a first cut at a class yielding NULL pointer objects
class NullClass {
public:
  template<class T>                        // generates
    operator T*() const { return 0; }      // operator T* for
};                                         // all types T; each
                                           // function returns
                                           // the null pointer


const NullClass NULL;             // NULL is an object of
                                  // type NullClass


void f(int x);                    // same as we originally had


void f(string *p);                // ditto


f(NULL);                          // fine, converts NULL to
                                  // string*, then calls f(string*)
```

This is a good initial draft, but it can be refined in several ways. First, we don't really need more

than one `NullClass` object, so there's no reason to give the class a name; we can just use an anonymous class and make `NULL` of that type. Second, as long as we're making it possible to convert `NULL` to any type of pointer, we should handle pointers to members, too. That calls for a second member template, one to convert `0` to type `T C::*` ("pointer to member of type `T` in class `C`") for all classes `C` and all types `T`. (If that makes no sense to you, or if you've never heard of — much less used — pointers to members, relax. Pointers to members are uncommon beasts, rarely seen in the wild, and you'll probably never have to deal with them. The terminally curious may wish to consult Item 30, which discusses pointers to members in a bit more detail.) Finally, we should prevent clients from taking the address of `NULL`, because `NULL` isn't supposed to act like a *pointer*, it's supposed to act like a pointer *value*, and pointer values (e.g., 0x453AB002) don't have addresses.

The jazzed-up `NULL` definition looks like this:

```
const                               // this is a const object...
class {
public:
  template<class T>                 // convertible to any type
    operator T*() const             // of null non-member
    { return 0; }                   // pointer...



  template<class C, class T>        // or any type of null
    operator T C::*() const         // member pointer...
    { return 0; }


private:
  void operator&() const;           // whose address can't be
                                    // taken (see Item 27)...



} NULL;                             // and whose name is NULL
```

This is truly a sight to behold, though you may wish to make a minor concession to practicality by giving the class a name after all. If you don't, compiler messages referring to `NULL`'s type are likely to be pretty unintelligible.

For another example of how member templates can be useful, take a look at Item M28.

An important point about all these attempts to come up with a workable `NULL` is that they help only if you're the *caller*. If you're the *author* of the functions being called, having a foolproof `NULL` won't

help you at all, because you can't compel your callers to use it. For example, even if you offer your clients the space-age `NULL` we just developed, you still can't keep them from doing this,

```
f(0);                   // still calls f(int),
                        // because 0 is still an int
```

and that's just as problematic now as it was at the beginning of this Item.

As a designer of overloaded functions, then, the bottom line is that you're best off avoiding overloading on a numerical and a pointer type if you can possibly avoid it.

Back to Item 24: Choose carefully between function overloading and parameter defaulting.
Continue to Item 26: Guard against potential ambiguity.

Back to Item 25: Avoid overloading on a pointer and a numerical type.
Continue to Item 27: Explicitly disallow use of implicitly generated member functions you don't want.

Item 26: Guard against potential ambiguity.

Everybody has to have a philosophy. Some people believe in *laissez faire* economics, others believe in reincarnation. Some people even believe that COBOL is a real programming language. C++ has a philosophy, too: it believes that potential ambiguity is not an error.

Here's an example of potential ambiguity:

```
class B;                    // forward declaration for
                            // class B
class A {
public:
  A(const B&);              // an A can be
                            // constructed from a B
};


class B {
public:
  operator A() const;       // a B can be
                            // converted to an A
};
```

There's nothing wrong with these class declarations — they can coexist in the same program without the slightest trouble. However, look what happens when you combine these classes with a function that takes an `A` object, but is actually passed a `B` object:

```
void f(const A&);


B b;



f(b);                       // error! – ambiguous
```

Seeing the call to `f`, compilers know they must somehow come up with an object of type `A`, even though what they have in hand is an object of type `B`. There are two equally good ways to do this (see Item M5). On one hand, the class `A` constructor could be called; this would construct a new `A`

object using `b` as an argument. On the other hand, `b` could be converted into an `A` object by calling the client-defined conversion operator in class `B`. Because these two approaches are considered equally good, compilers refuse to choose between them.

Of course, you could use this program for some time without ever running across the ambiguity. That's the insidious peril of potential ambiguity. It can lie dormant in a program for long periods of time, undetected and inactive, until the day when some unsuspecting programmer does something that actually *is* ambiguous, at which point pandemonium breaks out. This gives rise to the disconcerting possibility that you might release a library that can be called ambiguously without even being aware that you're doing it.

A similar form of ambiguity arises from standard conversions in the language — you don't even need any classes:

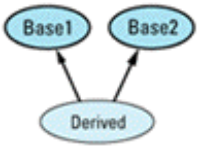
```
void f(int);
void f(char);


double d = 6.02;


f(d);                          // error! — ambiguous
```

Should `d` be converted into an `int` or a `char`? The conversions are equally good, so compilers won't judge. Fortunately, you can get around this problem by using an explicit cast:

```
f(static_cast<int>(d));        // fine, calls f(int)
f(static_cast<char>(d));       // fine, calls f(char)
```

Multiple inheritance (see Item 43) is rife with possibilities for potential ambiguity. The most straightforward case occurs when a derived class inherits the same member name from more than one base class:

```
class Base1 {
public:
  int doIt();
};
class Base2 {
public:
  void doIt();
};
```




```
class Derived: public Base1,          // Derived doesn't declare
               public Base2 {         // a function called doIt
  ...


};


Derived d;


d.doIt();                   // error! – ambiguous
```

When class `Derived` inherits two functions with the same name, C++ utters not a whimper; at this point the ambiguity is only potential. However, the call to `doIt` forces compilers to face the issue, and unless you explicitly disambiguate the call by specifying which base class function you want, the call is an error:

```
d.Base1::doIt();            // fine, calls Base1::doIt


d.Base2::doIt();            // fine, calls Base2::doIt
```

That doesn't upset too many people, but the fact that accessibility restrictions don't enter into the picture has caused more than one otherwise pacifistic soul to contemplate distinctly unpacifistic actions:

```
class Base1 { ... };        // same as above


class Base2 {
private:
  void doIt();              // this function is now
};                          // private
```

```
class Derived: public Base1, public Base2
{ ... };                    // same as above


Derived d;


int i = d.doIt();           // error! — still ambiguous!
```

The call to `doIt` continues to be ambiguous, even though only the function in `Base1` is accessible! The fact that only `Base1::doIt` returns a value that can be used to initialize an `int` is also irrelevant — the call remains ambiguous. If you want to make this call, you simply *must* specify which class's `doIt` is the one you want.

As is the case for most initially unintuitive rules in C++, there is a good reason why access restrictions are not taken into account when disambiguating references to multiply inherited members. It boils down to this: changing the accessibility of a class member should never change the meaning of a program.

For example, assume that in the previous example, access restrictions were taken into account. Then the expression `d.doIt()` would resolve to a call to `Base1::doIt`, because `Base2`'s version was inaccessible. Now assume that `Base1` was changed so that its version of `doIt` was protected instead of public, and `Base2` was changed so that its version was public instead of private.

Suddenly the same expression, `d.doIt()`, would result in a *completely different function call*, even though neither the calling code nor the functions had been modified! Now *that's* unintuitive, and there would be no way for compilers to issue even a warning. Considering your choices, you may decide that having to explicitly disambiguate references to multiply inherited members isn't quite as unreasonable as you originally thought.

Given that there are all these different ways to write programs and libraries harboring potential ambiguity, what's a good software developer to do? Primarily, you need to keep an eye out for it. It's next to impossible to root out all the sources of potential ambiguity, particularly when programmers combine libraries that were developed independently (see also Item 28), but by understanding the situations that often lead to potential ambiguity, you're in a better position to minimize its presence in the software you design and develop.

Back to Item 25: Avoid overloading on a pointer and a numerical type.
Continue to Item 27: Explicitly disallow use of implicitly generated member functions you don't want.

Back to Item 26: Guard against potential ambiguity.
Continue to Item 28: Partition the global namespace.

Item 27: Explicitly disallow use of implicitly generated member functions you don't want.

Suppose you want to write a class template, `Array`, whose generated classes behave like built-in C++ arrays in every way, except they perform bounds checking. One of the design problems you would face is how to prohibit assignment between `Array` objects, because assignment isn't legal for C++ arrays:

```
double values1[10];
double values2[10];


values1 = values2;                  // error!
```

For most functions, this wouldn't be a problem. If you didn't want to allow a function, you simply wouldn't put it in the class. However, the assignment operator is one of those distinguished member functions that C++, always the helpful servant, writes for you if you neglect to write it yourself (see Item 45). What then to do?

The solution is to declare the function, `operator=` in this case, *private*. By declaring a member function explicitly, you prevent compilers from generating their own version, and by making the function private, you keep people from calling it.

However, the scheme isn't foolproof; member and friend functions can still call your private function. *Unless*, that is, you are clever enough not to *define* the function. Then if you inadvertently call the function, you'll get an error at link-time (see Item 46).

For `Array`, your template definition would start out like this:

```
template<class T>
class Array {
private:
  // Don't define this function!
  Array& operator=(const Array& rhs);


  ...
```

```
};
```

Now if a client tries to perform assignments on `Array` objects, compilers will thwart the attempt, and if you inadvertently try it in a member or a friend function, the linker will yelp.

Don't assume from this example that this Item applies only to assignment operators. It doesn't. It applies to each of the compiler-generated functions described in Item 45. In practice, you'll find that the behavioral similarities between assignment and copy construction (see Items 11 and 16) almost always mean that anytime you want to disallow use of one, you'll want to disallow use of the other, too.

Back to Item 26: Guard against potential ambiguity.
Continue to Item 28: Partition the global namespace.

Back to Item 27: Explicitly disallow use of implicitly generated member functions you don't want.
Continue to Classes and Functions: Implementation

Item 28: Partition the global namespace.

The biggest problem with the global scope is that there's only one of them. In a large software project, there is usually a bevy of people putting names in this singular scope, and invariably this leads to name conflicts. For example, `library1.h` might define a number of constants, including the following:

```
const double LIB_VERSION = 1.204;
```

Ditto for `library2.h`:

```
const int LIB_VERSION = 3;
```

It doesn't take great insight to see that there is going to be a problem if a program tries to include both `library1.h` and `library2.h`. Unfortunately, outside of cursing under your breath, sending hate mail to the library authors, and editing the header files until the name conflicts are eliminated, there is little you can do about this kind of problem.

You can, however, take pity on the poor souls who'll have *your* libraries foisted on them. You probably already prepend some hopefully-unique prefix to each of your global symbols, but surely you must admit that the resulting identifiers are less than pleasing to gaze upon.

A better solution is to use a C++ `namespace`. Boiled down to its essence, a `namespace` is just a fancy way of letting you use the prefixes you know and love without making people look at them all the time. So instead of this,

```
const double sdmBOOK_VERSION = 2.0;      // in this library,
                                         // each symbol begins
class sdmHandle { ... };                 // with "sdm"

sdmHandle& sdmGetHandle();             // see Item 47 for why you
                                       // might want to declare
                                       // a function like this
```

you write this:

```
namespace sdm {
  const double BOOK_VERSION = 2.0;
  class Handle { ... };
```

```
  Handle& getHandle();
}
```

Clients then access symbols in your namespace in any of the usual three ways: by importing all the symbols in a namespace into a scope, by importing individual symbols into a scope, or by explicitly qualifying a symbol for one-time use. Here are some examples:

```
void f1()
{
  using namespace sdm;           // make all symbols in sdm
                                 // available w/o qualification
                                 // in this scope



  cout << BOOK_VERSION;          // okay, resolves to
                                 // sdm::BOOK_VERSION
  ...



  Handle h = getHandle();        // okay, Handle resolves to
                                 // sdm::Handle, getHandle
  ...                            // resolves to sdm::getHandle


}


void f2()
{
  using sdm::BOOK_VERSION;        // make only BOOK_VERSION
                                  // available w/o qualification
                                  // in this scope



  cout << BOOK_VERSION;           // okay, resolves to
                                  // sdm::BOOK_VERSION
  ...



  Handle h = getHandle();         // error! neither Handle
                                  // nor getHandle were
  ...                             // imported into this scope
```

```
}


void f3()
{
  cout << sdm::BOOK_VERSION;      // okay, makes BOOK_VERSION
                                  // available for this one use
  ...                             // only



  double d = BOOK_VERSION;        // error! BOOK_VERSION is
                                  // not in scope



  Handle h = getHandle();         // error! neither Handle
                                  // nor getHandle were
  ...                             // imported into this scope


}
```

(Some namespaces have no names. Such *unnamed namespaces* are used to limit the visibility of the elements inside the namespace. For details, see Item M31.)

One of the nicest things about namespaces is that potential ambiguity is not an error (see Item 26). As a result, you can import the same symbol from more than one namespace, yet still live a carefree life (provided you never actually use the symbol). For instance, if, in addition to namespace `sdm`, you had need to make use of this namespace,

```
namespace AcmeWindowSystem {


  ...


  typedef int Handle;


  ...


}
```

you could use both `sdm` and `AcmeWindowSystem` without conflict, provided you never referenced the symbol `Handle`. If you did refer to it, you'd have to explicitly say which namespace's `Handle` you wanted:

```
void f()
{
  using namespace sdm;                 // import sdm symbols
  using namespace AcmeWindowSystem;    // import Acme symbols



  ...                                  // freely refer to sdm
                                       // and Acme symbols
                                       // other than Handle



  Handle h;                            // error! which Handle?



  sdm::Handle h1;                      // fine, no ambiguity



  AcmeWindowSystem::Handle h2;         // also no ambiguity



  ...


}
```

Contrast this with the conventional header-file-based approach, where the mere inclusion of both `sdm.h` and `acme.h` would cause compilers to complain about multiple definitions of the symbol `Handle`.

Namespaces were added to C++ relatively late in the standardization game, so perhaps you think they're not that important and you can live without them. You can't. You can't, because almost everything in the standard library (see Item 49) lives inside the namespace `std`. That may strike you as a minor detail, but it affects you in a very direct manner: it's why C++ now sports funny-looking extensionless header names like `<iostream>`, `<string>`, etc. For details, turn to Item 49.

Because namespaces were introduced comparatively recently, your compilers might not yet support them. If that's the case, there's still no reason to pollute the global namespace, because you can approximate `namespaces` with `structs`. You do it by creating a struct to hold your global names, then putting your global names inside this struct as static members:

```
// definition of a struct emulating a namespace
struct sdm {
  static const double BOOK_VERSION;
  class Handle { ... };
  static Handle& getHandle();
};



const double sdm::BOOK_VERSION = 2.0;      // obligatory defn
                                           // of static data
                                           // member
```

Now when people want to access your global names, they simply prefix them with the struct name:

```
void f()
{
  cout << sdm::BOOK_VERSION;


  ...


  sdm::Handle h = sdm::getHandle();


  ...
}
```

If there are no name conflicts at the global level, clients of your library may find it cumbersome to use the fully qualified names. Fortunately, there is a way you can let them have their scopes and ignore them, too.

For your type names, provide typedefs that remove the need for explicit scoping. That is, for a type name `T` in your namespace-like struct `S`, provide a (global) typedef such that `T` is a synonym for `S::T`:

```
typedef sdm::Handle Handle;
```

For each (static) object `X` in your struct, provide a (global) reference `X` that is initialized with `S::X`:

```
const double& BOOK_VERSION = sdm::BOOK_VERSION;
```

Frankly, after you've read Item 47, the thought of defining a non-local static object like `BOOK_VERSION` will probably make you queasy. (You'll want to replace such objects with the functions described in Item 47.)

Functions are treated much like objects, but even though it's legal to define references to functions, future maintainers of your code will dislike you a lot less if you employ pointers to functions instead:

```
sdm::Handle& (* const getHandle)() =      // getHandle is a
  sdm::getHandle;                         // const pointer (see
                                          // Item 21) to
                                          // sdm::getHandle
```

Note that `getHandle` is a *const* pointer. You don't really want to let clients make it point to something other than `sdm::getHandle`, do you?

(If you're dying to know how to define a reference to a function, this should revitalize you:

```
sdm::Handle& (&getHandle)() =      // getHandle is a reference
  sdm::getHandle;                  // to sdm::getHandle
```

Personally, I think this is kind of cool, but there's a reason you've probably never seen this before. Except for how they're initialized, references to functions and const pointers to functions behave identically, and pointers to functions are much more readily understood.)

Given these typedefs and references, clients not suffering from global name conflicts can just use the unqualified type and object names, while clients who do have conflicts can ignore the typedef and reference definitions and use fully qualified names. It's unlikely that all your clients will want to use the shorthand names, so you should be sure to put the typedefs and references in a different header file from the one containing your `namespace`-emulating struct.

`structs` are a nice approximation to `namespaces`, but they're a long trek from the real thing. They fall short in a variety of ways, one of the most obvious of which is their treatment of operators.

Simply put, operators defined as `static` member functions of structs can be invoked only through a function call, never via the natural infix syntax that operators are designed to support:

```
// define a namespace-emulating struct containing
// types and functions for Widgets. Widget objects
// support addition via operator+
struct widgets {
  class Widget { ... };



  // see Item 21 for why the return value is const
  static const Widget operator+(const Widget& lhs,
                                const Widget& rhs);



  ...


};



// attempt to set up global (unqualified) names for
// Widget and operator+ as described above


typedef widgets::Widget Widget;



const Widget (* const operator+)(const Widget&,        // error!
                                 const Widget&);       // operator+
                                                       // can't be a
                                                       // pointer name



Widget w1, w2, sum;



sum = w1 + w2;                          // error! no operator+
                                        // taking Widgets is
                                        // declared at this
                                        // scope



sum = widgets::operator+(w1, w2);       // legal, but hardly
```

```
                                        // "natural" syntax
```

Such limitations should spur you to adopt real namespaces as soon as your compilers make it practical.

Back to Item 27: Explicitly disallow use of implicitly generated member functions you don't want.
Continue to Classes and Functions: Implementation

Back to Item 28: Partition the global namespace.
Continue to Item 29: Avoid returning "handles" to internal data.

Classes and Functions: Implementation

Because C++ is strongly typed, coming up with appropriate definitions for your classes and templates and appropriate declarations for your functions is the lion's share of the battle. Once you've got those right, it's hard to go wrong with the template, class, and function implementations. Yet, somehow, people manage to do it.

Some problems arise from inadvertently violating abstraction: accidentally allowing implementation details to peek out from behind the class and function boundaries that are supposed to contain them. Others originate in confusion over the length of an object's lifetime. Still others stem from premature optimization, typically traceable to the seductive nature of the `inline` keyword. Finally, some implementation strategies, while fine on a local scale, result in levels of coupling between source files that can make it unacceptably costly to rebuild large systems.

Each of these problems, as well as others like them, can be avoided if you know what to watch out for. The items that follow identify some situations in which you need to be especially vigilant.

Back to Item 28: Partition the global namespace.
Continue to Item 29: Avoid returning "handles" to internal data.

Back to Implementation
Continue to Item 30: Avoid member functions that return non-const pointers or references to members less accessible than themselves.

Item 29: Avoid returning "handles" to internal data.

A scene from an object-oriented romance:

> Object A: Darling, don't ever change!
>
> Object B: Don't worry, dear, I'm `const`.

Yet just as in real life, A wonders, "Can B be trusted?" And just as in real life, the answer often hinges on B's nature: the constitution of its member functions.

Suppose B is a constant `String` object:

```
class String {
public:
  String(const char *value);        // see Item 11 for pos-
  ~String();                        // sible implementations;
                                    // see Item M5 for comments
                                    // on the first constructor



  operator char *() const;          // convert String -> char*;
                                    // see also Item M5
  ...

private:
  char *data;
};



const String B("Hello World");      // B is a const object
```

Because `B` is `const`, it had better be the case that the value of `B` now and evermore is "Hello World". Of course, this supposes that programmers working with `B` are playing the game in a civilized fashion. In particular, it depends on the fact that nobody is "casting away the constness" of `B` through nefarious ploys such as this (see Item 21):

```
String& alsoB =             // make alsoB another name
```

```
  const_cast<String&>(B);    // for B, but without the
                             // constness
```

Given that no one is doing such evil deeds, however, it seems a safe bet that `B` will never change. Or does it? Consider this sequence of events:

```
char *str = B;               // calls B.operator char*()


strcpy(str, "Hi Mom");       // modifies what str
                             // points to
```

Does `B` still have the value "Hello World", or has it suddenly mutated into something you might say to your mother? The answer depends entirely on the implementation of `String::operator char*`.

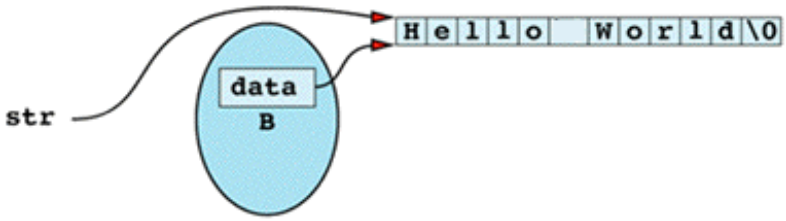
Here's a careless implementation, one that does the wrong thing. However, it does it very efficiently, which is why so many programmers fall into this trap:

```
// a fast, but incorrect implementation
inline String::operator char*() const
{ return data; }
```

The flaw in this function is that it's returning a "handle" — in this case, a pointer — to information that should be hidden inside the `String` object on which the function is invoked. That handle gives callers unrestricted access to what the private field `data` points to. In other words, after the statement

```
char *str = B;
```

the situation looks like this:



Clearly, any modification to the memory pointed to by `str` will also change the effective value of `B`. Thus, even though `B` is declared `const`, and even though only `const` member functions are invoked on `B`, `B` might still acquire different values as the program runs. In particular, if `str` modifies what it points to, `B` will also change.

There's nothing inherently wrong with the way `String::operator char*` is written. What's troublesome is that it can be applied to constant objects. If the function weren't declared `const`, there would be no problem, because it couldn't be applied to objects like `B`.

Yet it seems perfectly reasonable to turn a `String` object, even a constant one, into its equivalent `char*`, so you'd like to keep this function `const`. If you want to do that, you must rewrite your implementation to avoid returning a handle to the object's internal data:

```
// a slower, but safer implementation
inline String::operator char*() const
{
  char *copy = new char[strlen(data) + 1];
  strcpy(copy, data);


  return copy;


}
```

This implementation is safe, because it returns a pointer to memory that contains a *copy* of the data to which the `String` object points; there is no way to change the value of the `String` object through the pointer returned by this function. As usual, such safety commands a price: this version of `String::operator char*` is slower than the simple version above, and callers of this function must remember to use `delete` on the pointer that's returned.

If you think this version of `operator char*` is too slow, or if the potential memory leak makes you

nervous (as well it should), a slightly different tack is to return a pointer to *constant* `char`s:

```
class String {
public:
  operator const char *() const;


  ...

};


inline String::operator const char*() const
{ return data; }
```

This function is fast and safe, and, though it's not the same as the function you originally specified, it suffices for most applications. It's also the moral equivalent of the ◦C++ standardization committee's solution to the `string`/`char*` conundrum: the standard `string` type contains a member function `c_str` that returns a `const char*` version of the `string` in question. For more information on the standard `string` type, turn to Item 49.

A pointer isn't the only way to return a handle to internal data. References are just as easy to abuse. Here's a common way to do it, again using the `String` class:

```
class String {
public:


  ...


  char& operator[](int index) const
  { return data[index]; }


private:
  char *data;
};


String s = "I'm not constant";



s[0] = 'x';               // fine, s isn't const
```

```
const String cs = "I'm constant";



cs[0] = 'x';              // this modifies the const
                          // string, but compilers
                          // won't notice
```

Notice how `String::operator[]` returns its result by reference. That means that the caller of this function gets back *another name* for the internal element `data[index]`, and that other name can be used to modify the internal data of the supposedly constant object. This is the same problem you saw before, but this time the culprit is a reference as a return value, not a pointer.

The general solutions to this kind of problem are the same as they were for pointers: either make the function non-`const`, or rewrite it so that no handle is returned. For a solution to this *particular* problem — how to write `String::operator[]` so that it works for both `const` and non-`const` objects — see Item 21.

`const` member functions aren't the only ones that need to worry about returning handles. Even non-`const` member functions must reconcile themselves to the fact that the validity of a handle expires at the same time as the object to which it corresponds. This may be sooner than a client expects, especially when the object in question is a compiler-generated temporary object.

For example, take a look at this function, which returns a `String` object:

```
String someFamousAuthor()           // randomly chooses and
{                                   // returns an author's name



  switch (rand() % 3) {             // rand() is in <stdlib.h>
                                    // (and <cstdlib> — see
                                    // Item 49)
  case 0:
    return "Margaret Mitchell";     // Wrote "Gone with the
                                    // Wind," a true classic
  case 1:
    return "Stephen King";          // His stories have kept
                                    // millions from sleeping
                                    // at night
  case 2:
    return "Scott Meyers";          // Ahem, one of these
  }                                 // things is not like the
```

```
                                    // others...


  return "";                        // we can't get here, but
                                    // all paths in a value-
                                    // returning function must
}                                   // return a value, sigh
```

Kindly set aside your concerns about how "random" the values returned from `rand` are, and please humor my delusions of grandeur in associating myself with real writers. Instead, focus on the fact that the return value of `someFamousAuthor` is a `String` object, a *temporary* `String` object (see Item M19). Such objects are transient — their lifetimes generally extend only until the end of the expression containing the call to the function creating them. In this case, that would be until the end of the expression containing the call to `someFamousAuthor`.

Now consider this use of `someFamousAuthor`, in which we assume that `String` declares an `operator const char*` member function as described above:

```
const char *pc = someFamousAuthor();


cout << pc;                     // uh oh...
```

Believe it or not, you can't predict what this code will do, at least not with any certainty. That's because by the time you try to print out the sequence of characters pointed to by `pc`, that sequence is undefined. The difficulty arises from the events that transpire during the initialization of `pc`:

1. A temporary `String` object is created to hold `someFamousAuthor`'s return value.
2. That `String` is converted to a `const char*` via `String`'s `operator const char*` member function, and `pc` is initialized with the resulting pointer.
3. The temporary `String` object is destroyed, which means its destructor is called. Within the destructor, its `data` pointer is deleted (the code is shown in Item 11). However, `data` points to the same memory as `pc` does, so `pc` now points to deleted memory — memory with undefined contents.

Because `pc` was initialized with a handle into a temporary object and temporary objects are destroyed shortly after they're created, the handle became invalid before `pc` could do anything with it. For all intents and purposes, `pc` was dead on arrival. Such is the danger of handles into temporary objects.

For `const` member functions, then, returning handles is ill-advised, because it violates abstraction.

Even for non-`const` member functions, however, returning handles can lead to trouble, especially when temporary objects get involved. Handles can dangle, just like pointers, and just as you labor to avoid dangling pointers, you should strive to avoid dangling handles, too.

Still, there's no reason to get fascist about it. It's not possible to stomp out all possible dangling pointers in nontrivial programs, and it's rarely possible to eliminate all possible dangling handles, either. Nevertheless, if you avoid returning handles when there's no compelling need, your programs will benefit, and so will your reputation.

Back to Implementation
Continue to Item 30: Avoid member functions that return non-const pointers or references to members less accessible than themselves.

Back to Item 29: Avoid returning "handles" to internal data.
Continue to Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.

Item 30: Avoid member functions that return non-`const` pointers or references to members less accessible than themselves.

The reason for making a member private or protected is to limit access to it, right? Your overworked, underpaid C++ compilers go to lots of trouble to make sure that your access restrictions aren't circumvented, right? So it doesn't make a lot of sense for you to write functions that give random clients the ability to freely access restricted members, now, does it? If you think it *does* make sense, please reread this paragraph over and over until you agree that it doesn't. It's easy to violate this simple rule. Here's an example:

```
class Address { ... };           // where someone lives

class Person {
public:
  Address& personAddress() { return address; }
  ...

private:
  Address address;
  ...
};
```

The member function `personAddress` provides the caller with the `Address` object contained in the `Person` object, but, probably due to efficiency considerations, the result is returned by reference instead of by value (see Item 22). Unfortunately, the presence of this member function defeats the purpose of making `Person::address` private:

```
Person scott(...);             // parameters omitted for
                               // simplicity

Address& addr =                // assume that addr is
  scott.personAddress();       // global
```

Now the global object `addr` is *another name* for `scott.address`, and it can be used to read and write `scott.address` at will. For all practical purposes, `scott.address` is no longer private; it is public, and the source of this promotion in accessibility is the member function `personAddress`. Of course, there is nothing special about the access level `private` in this example; if `address` were protected, exactly the same reasoning would apply. References aren't the only cause for concern. Pointers can play this game, too. Here's the same example, but using pointers this time:

```
class Person {
public:
  Address * personAddress() { return &address; }
  ...
```

```
private:
  Address address;
  ...
};

Address *addrPtr =
  scott.personAddress();        // same problem as above
```

With pointers, however, you have to worry not only about data members, but also about member *functions*. That's because it's possible to return a pointer to a member function:

```
class Person;                   // forward declaration

// PPMF = "pointer to Person member function"
typedef void (Person::*PPMF)();

class Person {
public:
  static PPMF verificationFunction()
  { return &Person::verifyAddress; }

  ...

private:
  Address address;

  void verifyAddress();

};
```

If you're not used to socializing with pointers to member functions and typedefs thereof, the declaration for `Person::verificationFunction` may seem daunting. Don't be intimidated. All it says is

- `verificationFunction` is a member function that takes no parameters;
- its return value is a pointer to a member function of the `Person` class;
- the pointed-to function (i.e., `verificationFunction`'s return value) takes no parameters and returns nothing, i.e., `void`.

As for the word `static`, that means what it always means in a member declaration: there is only one copy of the member for the entire class, and the member can be accessed without an object. For the complete story, consult your favorite introductory C++ textbook. (If your favorite introductory C++ textbook doesn't discuss static members, carefully tear out all its pages and recycle them. Dispose of the book's cover in an environmentally sound manner, then borrow or buy a better textbook.) In this last example, `verifyAddress` is a private member function, indicating that it's really an implementation detail of the class; only class members should know about it (and friends, too, of course). However, the public member function `verificationFunction` returns a pointer to `verifyAddress`, so clients can again pull this kind of thing:

```
PPMF pmf = scott.verificationFunction();

(scott.*pmf)();                     // same as calling
                                    // scott.verifyAddress
```

Here, `pmf` has become a synonym for `Person::verifyAddress`, with the crucial difference that there are no restrictions on its use. In spite of the foregoing discussion, you may someday be faced with a situation in which, pressed to achieve performance constraints, you honestly need to write a member function that returns a reference or a pointer to a less-accessible member. At the same time, however, you won't want to sacrifice the access restrictions that `private` and `protected` afford you. In those cases, you can almost always achieve both goals by returning a pointer or a reference to a `const` object. For details, take a look at Item 21.

Back to Item 29: Avoid returning "handles" to internal data.
Continue to Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.

Back to Item 30: Avoid member functions that return non-const pointers or references to members less accessible than themselves.
Continue to Item 32: Postpone variable definitions as long as possible.

Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by **new** within the function.

This Item may sound complicated, but it's not. It's simple common sense. Really. Honest. *Trust me.*

Consider first the matter of returning a reference to a local object. The problem here is that local objects are just that, *local*. That means they're constructed when they're defined, and they're destructed when they go out of scope. Their scope, however, is that of the function body in which they're located. When the function returns, control leaves its scope, so the objects local to that function are automatically destructed. As a result, if you return a reference to a local object, that local object has been destructed before the caller of the function ever gets its computational hands on it.

This problem usually raises its ugly head when you try to improve the efficiency of a function by returning its result by reference instead of by value. The following example is the same as the one in Item 23, which pursues in detail the question of when you can return a reference and when you can't:

```
class Rational {          // class for rational numbers
public:
  Rational(int numerator = 0, int denominator = 1);
  ~Rational();


  ...


private:
  int n, d;               // numerator and denominator


// notice that operator* (incorrectly) returns a reference
friend const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs);
};


// an incorrect implementation of operator*
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
```

```
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

Here, the local object `result` is constructed upon entry into the body of `operator*`. However, local objects are automatically destroyed when they go out of scope. `result` will go out of scope after execution of the `return` statement, so when you write this,

```
Rational two = 2;



Rational four = two * two;          // same as
                                    // operator*(two, two)
```

what happens during the function call is this:

1. The local object `result` is constructed.
2. A reference is initialized to be another name for `result`, and this reference is squirreled away as `operator*`'s return value.
3. The local object `result` is destroyed, and the space it used to occupy on the stack is made available for use by other parts of the program or by other programs.
4. The object `four` is initialized using the reference of step 2.

Everything is fine until step 4, at which point there occurs, as they say in the highest of high-tech circles, "a major lossage." The reference initialized in step 2 ceased to refer to a valid object as of the end of step 3, so the outcome of the initialization of object `four` is completely undefined.

The lesson should be clear: don't return a reference to a local object.

"Okay," you say, "the problem is that the object I want to use goes out of scope too soon. I can fix that. I'll just call `new` instead of using a local object." Like this:

```
// another incorrect implementation of operator*
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  // create a new object on the heap
  Rational *result =
    new Rational(lhs.n * rhs.n, lhs.d * rhs.d);



  // return it
  return *result;
```

```
}
```

This approach does indeed avoid the problem of the previous example, but it introduces a new one in its place. To avoid a memory leak in your software, you know you must ensure that `delete` is applied to every pointer conjured up by `new`, but ay, there's the rub: who's to make the matching call to `delete` for this function's use of `new`?

Clearly, the *caller* of `operator*` must see to it that `delete` is applied. Clear, yes, and even easy to document, but nonetheless the cause is hopeless. There are two reasons for this pessimistic assessment.

First, it's well-known that programmers, as a breed, are sloppy. That doesn't mean that you're sloppy or that I'm sloppy, but rare is the programmer who doesn't work with someone who is — shall we say? — a little on the flaky side. What are the odds that such programmers — and we all know that they exist — will remember that whenever they call `operator*`, they must *take the address of the result* and then use `delete` on it? That is, they must use `operator*` like this:

```
const Rational& four = two * two;      // get dereferenced
                                       // pointer; store it in
                                       // a reference
...



delete &four;                          // retrieve pointer
                                       // and delete it
```

The odds are vanishingly small. Remember, if only a *single caller* of `operator*` fails to follow the rules, you have a memory leak.

Returning dereferenced pointers has a second, more serious, problem, because it persists even in the presence of the most conscientious of programmers. Often, the result of `operator*` is a temporary intermediate value, an object that exists only for the purposes of evaluating a larger expression. For example:

```
Rational one(1), two(2), three(3), four(4);
Rational product;



product = one * two * three * four;
```

Evaluation of the expression to be assigned to `product` requires three separate calls to `operator*`, a fact that becomes more evident when you rewrite the expression in its equivalent functional form:

```
product = operator*(operator*(operator*(one, two), three), four);
```

You know that each of the calls to `operator*` returns an object that needs to be deleted, but there is no possibility of applying `delete`, because none of the returned objects has been saved anywhere.

The only solution to this difficulty is to ask clients to code like this:

```
const Rational& temp1 = one * two;
const Rational& temp2 = temp1 * three;
const Rational& temp3 = temp2 * four;



delete &temp1;
delete &temp2;
delete &temp3;
```

Do that, and the best you can hope for is that people will ignore you. More realistically, you'd be skinned alive, or possibly sentenced to ten years hard labor writing microcode for waffle irons and toaster ovens.

Learn your lesson now, then: writing a function that returns a dereferenced pointer is a memory leak just waiting to happen.

By the way, if you think you've come up with a way to avoid the undefined behavior inherent in returning a reference to a local object and the memory leak haunting the return of a reference to a heap-allocated object, turn to Item 23 and read why returning a reference to a local `static` object also fails to work correctly. It may save you the trouble of seeking medical care for the arm you're likely to strain trying to pat yourself on the back.

Back to Item 30: Avoid member functions that return non-const pointers or references to members less accessible than themselves.
Continue to Item 32: Postpone variable definitions as long as possible.

Back to Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.
Continue to Item 33: Use inlining judiciously.

Item 32: Postpone variable definitions as long as possible.

So you subscribe to the C philosophy that variables should be defined at the beginning of a block. Cancel that subscription! In C++, it's unnecessary, unnatural, and expensive.

Remember that when you define a variable of a type with a constructor or destructor, you incur the cost of construction when control reaches the variable's definition, and you incur the cost of destruction when the variable goes out of scope. This means there's a cost associated with unused variables, so you want to avoid them whenever you can.

Suave and sophisticated in the ways of programming as I know you to be, you're probably thinking you never define unused variables, so this Item's advice is inapplicable to your tight, lean coding style. You may need to think again. Consider the following function, which returns an encrypted version of a password, provided the password is long enough. If the password is too short, the function throws an exception of type `logic_error`, which is defined in the standard C++ library (see Item 49):

```
// this function defines the variable "encrypted" too soon
string encryptPassword(const string& password)
{
  string encrypted;


  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
     throw logic_error("Password is too short");
  }


  do whatever is necessary to place an encrypted
  version of password in encrypted;


  return encrypted;
}
```

The object `encrypted` isn't *completely* unused in this function, but it's unused if an exception is thrown. That is, you'll pay for the construction and destruction of `encrypted` even if `encryptPassword` throws an exception (see also Item M15). As a result, you're better off postponing `encrypted`'s definition until you *know* you'll need it:

```
// this function postpones "encrypted"'s definition until
// it's truly necessary
string encryptPassword(const string& password)
{
  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }


  string encrypted;


  do whatever is necessary to place an encrypted
  version of password in encrypted;


  return encrypted;
}
```

This code still isn't as tight as it might be, because `encrypted` is defined without any initialization arguments. That means its default constructor will be used. In many cases, the first thing you'll do to an object is give it some value, often via an assignment. Item 12 explains why default-constructing an object and then assigning to it is a lot less efficient than initializing it with the value you really want it to have. That analysis applies here, too. For example, suppose the hard part of `encryptPassword` is performed in this function:

```
void encrypt(string& s);      // encrypts s in place
```

Then `encryptPassword` could be implemented like this, though it wouldn't be the best way to do it:

```
// this function postpones "encrypted"'s definition until
// it's necessary, but it's still needlessly inefficient
string encryptPassword(const string& password)
{
  ...                     // check length as above


  string encrypted;       // default-construct encrypted
  encrypted = password;   // assign to encrypted
  encrypt(encrypted);
  return encrypted;
}
```

A preferable approach is to initialize `encrypted` with `password`, thus skipping the (pointless) default construction:

```
// finally, the best way to define and initialize encrypted
string encryptPassword(const string& password)
{
  ...                               // check length


  string encrypted(password);       // define and initialize
                                    // via copy constructor


  encrypt(encrypted);
  return encrypted;
}
```

This suggests the real meaning of "as long as possible" in this Item's title. Not only should you postpone a variable's definition until right before you have to use the variable, you should try to postpone the definition until you have initialization arguments for it. By doing so, you avoid not only constructing and destructing unneeded objects, you also avoid pointless default constructions. Further, you help document the purpose of variables by initializing them in contexts in which their meaning is clear. Remember how in C you're encouraged to put a short comment after each variable definition to explain what the variable will eventually be used for? Well, combine decent variable names (see also Item 28) with contextually meaningful initialization arguments, and you have every programmer's dream: a solid argument for *eliminating* some comments.

By postponing variable definitions, you improve program efficiency, increase program clarity, and reduce the need to document variable meanings. It looks like it's time to kiss those block-opening variable definitions good-bye.

Back to Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.
Continue to Item 33: Use inlining judiciously.

Back to Item 32: Postpone variable definitions as long as possible.
Continue to Item 34: Minimize compilation dependencies between files.

Item 33: Use inlining judiciously.

Inline functions -- what a *wonderful* idea! They look like functions, they act like functions, they're ever so much better than macros (see Item 1), and you can call them without having to incur the overhead of a function call. What more could you possibly ask for?

You actually get more than you might think, because avoiding the cost of a function call is only half the story. Compiler optimization routines are typically designed to concentrate on stretches of code that lack function calls, so when you inline a function, you may enable compilers to perform context-specific optimizations on the body of the function. Such optimizations would be impossible for "normal" function calls.

However, let's not get carried away. In programming, as in life, there is no free lunch, and inline functions are no exception. The whole idea behind an inline function is to replace each call of that function with its code body, and it doesn't take a Ph.D. in statistics to see that this is likely to increase the overall size of your object code. On machines with limited memory, overzealous inlining can give rise to programs that are too big for the available space. Even with virtual memory, inline-induced code bloat can lead to pathological paging behavior (thrashing) that will slow your program to a crawl. (It will, however, provide your disk controller with a nice exercise regimen.) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

On the other hand, if an inline function body is *very* short, the code generated for the function body may actually be smaller than the code generated for a function call. If that is the case, inlining the function may actually lead to *smaller* object code and a higher cache hit rate!

Bear in mind that the `inline` directive, like `register`, is a *hint* to compilers, not a command. That means compilers are free to ignore your inline directives whenever they want to, and it's not that hard to make them want to. For example, most compilers refuse to inline "complicated" functions (e.g., those that contain loops or are recursive), and all but the most trivial virtual function calls stop inlining routines dead in their tracks. (This shouldn't be much of a surprise. `virtual` means "wait until runtime to figure out which function to call," and `inline` means "during compilation, replace the call site with the called function." If compilers don't know which function will be called, you can hardly blame them for refusing to make an inline call to it.) It all adds up to this: whether a given inline function is actually inlined is dependent on the implementation of the compiler you're using. Fortunately, most compilers have a diagnostic level that will result in a warning (see Item 48) if they fail to inline a function you've asked them to.

Suppose you've written some function `f` and you've declared it `inline`. What happens if a compiler

chooses, for whatever reason, not to inline that function? The obvious answer is that `f` will be treated like a non-inline function: code for `f` will be generated as if it were a normal "outlined" function, and calls to `f` will proceed as normal function calls.

In theory, this is precisely what will happen, but this is one of those occasions when theory and practice may go their separate ways. That's because this very tidy solution to the problem of what to do about "outlined inlines" was added to C++ relatively late in the standardization process. Earlier specifications for the language (such as the ARM — see Item 50) told compiler vendors to implement different behavior, and the older behavior is still common enough that you need to understand what it is.

Think about it for a minute, and you'll realize that inline function definitions are virtually always put in header files. This allows multiple translation units (source files) to include the same header files and reap the advantages of the inline functions that are defined within them. Here's an example, in which I adopt the convention that source files end in ".cpp"; this is probably the most prevalent of the file naming conventions in the world of C++:

```
// This is file example.h
inline void f() { ... }          // definition of f

...


// This is file source1.cpp
#include "example.h"             // includes definition of f

...                              // contains calls to f


// This is file source2.cpp
#include "example.h"             // also includes definition
                                 // of f

...                              // also calls f
```

Under the old "outlined inline" rules and the assumption that `f` is *not* being inlined, when `source1.cpp` is compiled, the resulting object file will contain a function called `f`, just as if `f` had never been declared `inline`. Similarly, when `source2.cpp` is compiled, its generated object file will also hold a function called `f`. When you try to link the two object files together, you can reasonably expect your linker to complain that your program contains two definitions of `f`, an error.

To prevent this problem, the old rules decreed that compilers treat an un-inlined inline function as if the function had been declared `static` — that is, local to the file currently being compiled. In the example you just saw, compilers following the old rules would treat `f` as if it were static in

source1.cpp when that file was being compiled and as if it were static in source2.cpp when that file was being compiled. This strategy eliminates the link-time problem, but at a cost: each translation unit that includes the definition of f (and that calls f) contains its own static copy of f. If f itself defines local static variables, each copy of f gets its *own copy* of the variables, something sure to astonish programmers who believe that "static" inside a function means "only one copy."

This leads to a stunning realization. Under both new rules and old, if an inline function isn't inlined, you *still* pay for the cost of a function call at each call site, but under the old rules, you can *also* suffer an increase in code size, because each translation unit that includes and calls f gets its own copy of f's code and f's static variables! (To make matters worse, each copy of f and each copy of f's static variables tend to end up on different virtual memory pages, so two calls to different copies of f are likely to entail one or more page faults.)

There's more. Sometimes your poor, embattled compilers have to generate a function body for an inline function even when they are perfectly willing to inline the function. In particular, if your program ever takes the address of an inline function, compilers must generate a function body for it. How can they come up with a pointer to a function that doesn't exist?

```
inline void f() {...}            // as above

void (*pf)() = f;                // pf points to f

int main()
{
  f();                           // an inline call to f

  pf();                          // a non-inline call to f
                                 // through pf
  ...
}
```

In this case, you end up in the seemingly paradoxical situation whereby calls to f are inlined, but — under the old rules — each translation unit that takes f's address still generates a static copy of the function. (Under the new rules, only a single out-of-line copy of f will be generated, regardless of the number of translation units involved.)

This aspect of un-inlined inline functions can affect you even if you never use function pointers, because programmers aren't necessarily the only ones asking for pointers to functions. Sometimes compilers do it. In particular, compilers sometimes generate out-of-line copies of constructors and destructors so that they can get pointers to those functions for use in constructing and destructing arrays of objects of a class (see also Item M8).

In fact, constructors and destructors are often worse candidates for inlining than a casual examination would indicate. For example, consider the constructor for class Derived below:

```
class Base {
public:
  ...

private:
  string bm1, bm2;   // base members 1 and 2
};

class Derived: public Base {
public:
  Derived() {}                  // Derived's ctor is
  ...                           // empty -- or is it?



private:
  string dm1, dm2, dm3;         // derived members 1-3
};
```

This constructor certainly looks like an excellent candidate for inlining, since it contains no code. But looks can be deceiving. Just because it contains no code doesn't necessarily mean it contains no code. In fact, it may contain a fair amount of code.

C++ makes various guarantees about things that happen when objects are created and destroyed. Items 5 and M8 describes how when you use `new`, your dynamically created objects are automatically initialized by their constructors, and how when you use `delete`, the corresponding destructors are invoked. Item 13 explains that when you create an object, each base class of and each data member in that object is automatically constructed, and the reverse process regarding destruction automatically occurs when an object is destroyed. Those items describe what C++ says must happen, but C++ does not say *how* they happen. That's up to compiler implementers, but it should be clear that those things don't just happen by themselves. There has to be some code in your program to make those things happen, and that code — the code written by compiler implementers and inserted into your program during compilation — has to go somewhere. Sometimes, it ends up in your constructors and destructors, so some implementations will generate code equivalent to the following for the allegedly empty `Derived` constructor above:

```
// possible implementation of Derived constructor
 Derived::Derived()
{
  // allocate heap memory for this object if it's supposed
  // to be on the heap; see Item 8 for info on operator new
  if (this object is on the heap)
    this = ::operator new(sizeof(Derived));



  Base::Base();                  // initialize Base part

  dm1.string();             // construct dm1
```

```
  dm2.string();            // construct dm2
  dm3.string();            // construct dm3
}
```

You could never hope to get code like this to compile, because it's not legal C++ — not for you, anyway. For one thing, you have no way of asking whether an object is on the heap from inside its constructor. (For an examination of what it takes to reliably determine whether an object is on the heap, see Item M27.) For another, you're forbidden from assigning to `this`. And you can't invoke constructors via function calls, either. Your compilers, however, labor under no such constraints — they can do whatever they like. But the legality of the code is not the point. The point is that code to call `operator new` (if necessary), to construct base class parts, and to construct data members may be silently inserted into your constructors, and when it is, those constructors increase in size, thus making them less attractive candidates for inlining. Of course, the same reasoning applies to the `Base` constructor, so if it's inlined, all the code inserted into it is also inserted into the `Derived` constructor (via the `Derived` constructor's call to the `Base` constructor). And if the `string` constructor also happens to be inlined, the `Derived` constructor will gain *five copies* of that function's code, one for each of the five strings in a `Derived` object (the two it inherits plus the three it declares itself). Now do you see why it's not necessarily a no-brain decision whether to inline `Derived`'s constructor? Of course, similar considerations apply to `Derived`'s destructor, which, one way or another, must see to it that all the objects initialized by `Derived`'s constructor are properly destroyed. It may also need to free the dynamically allocated memory formerly occupied by the just-destroyed `Derived` object.

Library designers must evaluate the impact of declaring functions `inline`, because inline functions make it impossible to provide binary upgrades to the inline functions in a library. In other words, if `f` is an inline function in a library, clients of the library compile the body of `f` into their applications. If a library implementer later decides to change `f`, all clients who've used `f` must recompile. This is often highly undesirable (see also Item 34). On the other hand, if `f` is a non-`inline` function, a modification to `f` requires only that clients relink. This is a substantially less onerous burden than recompiling and, if the library containing the function is dynamically linked, one that may be absorbed in a way that's completely transparent to clients.

Static objects inside inline functions often exhibit counterintuitive behavior. For this reason, it's generally a good idea to avoid declaring functions `inline` if those functions contain static objects. For details, consult Item M26.

For purposes of program development, it is important to keep all these considerations in mind, but from a purely practical point of view during coding, one fact dominates all others: most debuggers have trouble with inline functions.

This should be no great revelation. How do you set a breakpoint in a function that isn't there? How do you step through such a function? How do you trap calls to it? Without being unreasonably

clever (or deviously underhanded), you simply can't. Happily, this leads to a logical strategy for determining which functions should be declared `inline` and which should not.

Initially, don't inline anything, or at least limit your inlining to those functions that are truly trivial, such as `age` below:

```
class Person {
public:
  int age() const { return personAge; }

  ...

private:
  int personAge;

  ...

};
```

By employing inlines cautiously, you facilitate your use of a debugger, but you also put inlining in its proper place: as a hand-applied optimization. Don't forget the empirically determined rule of 80-20 (see Item M16), which states that a typical program spends 80 percent of its time executing only 20 percent of its code. It's an important rule, because it reminds you that your goal as a software developer is to identify the 20 percent of your code that is actually capable of increasing your program's overall performance. You can inline and otherwise tweak your functions until the cows come home, but it's all wasted effort unless you're focusing on the *right* functions.

Once you've identified the set of important functions in your application, the ones whose inlining will actually make a difference (a set that is itself dependent on the architecture on which you're running), don't hesitate to declare them `inline`. At the same time, however, be on the lookout for problems caused by code bloat, and watch out for compiler warnings (see Item 48) that indicate that your inline functions haven't been inlined.

Used judiciously, inline functions are an invaluable component of every C++ programmer's toolbox, but, as the foregoing discussion has revealed, they're not quite as simple and straightforward as you might have thought.

Back to Item 32: Postpone variable definitions as long as possible.
Continue to Item 34: Minimize compilation dependencies between files.

Back to Item 33: Use inlining judiciously.
Continue to Inheritance and Object-Oriented Design

Item 34: Minimize compilation dependencies between files.

So you go into your C++ program and you make a minor change to the implementation of a class. Not the class interface, mind you, just the implementation; only the private stuff. Then you get set to rebuild the program, figuring that the compilation and linking should take only a few seconds. After all, only one class has been modified. You click on Rebuild or type `make` (or its moral equivalent), and you are astonished, then mortified, as you realize that the whole *world* is being recompiled and relinked!

Don't you just *hate* it when that happens?

The problem is that C++ doesn't do a very good job of separating interfaces from implementations. In particular, class definitions include not only the interface specification, but also a fair number of implementation details. For example:

```
class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();



  ...                       // copy constructor and assignment
                            // operator omitted for simplicity
  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;



private:
  string name_;             // implementation detail
  Date birthDate_;          // implementation detail
  Address address_;         // implementation detail
  Country citizenship_;     // implementation detail
};
```

This is hardly a Nobel Prize-winning class design, although it does illustrate an interesting naming convention for distinguishing private data from public functions when the same name makes sense for both: the former are tagged with a trailing underbar. The important thing to observe is that class `Person` can't be compiled unless the compiler also has access to definitions for the classes in terms

of which `Person` is implemented, namely, `string`, `Date`, `Address`, and `Country`. Such definitions are typically provided through `#include` directives, so at the top of the file defining the `Person` class, you are likely to find something like this:

```
#include <string>           // for type string (see Item 49)
#include "date.h"
#include "address.h"
#include "country.h"
```

Unfortunately, this sets up a compilation dependency between the file defining `Person` and these include files. As a result, if any of these auxiliary classes changes its implementation, or if any of the classes on which it depends changes *its* implementation, the file containing the `Person` class must be recompiled, as must any files that use the `Person` class. For clients of `Person`, this can be more than annoying. It can be downright incapacitating.

You might wonder why C++ insists on putting the implementation details of a class in the class definition. For example, why can't you define `Person` this way,

```
class string;         // "conceptual" forward declaration for the
                      // string type. See Item 49 for details.



class Date;           // forward declaration
class Address;        // forward declaration
class Country;        // forward declaration


class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();


  ...                      // copy ctor, operator=


  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;
};
```

specifying the implementation details of the class separately? If that were possible, clients of `Person` would have to recompile only if the interface to the class changed. Because interfaces tend to stabilize before implementations do, such a separation of interface from implementation could save untold hours of recompilation and linking over the course of a large software effort.

Alas, the real world intrudes on this idyllic scenario, as you will appreciate when you consider something like this:

```
int main()
{
  int x;                      // define an int



  Person p(...);              // define a Person
                              // (arguments omitted for
  ...                         // simplicity)


}
```

When compilers see the definition for `x`, they know they must allocate enough space to hold an `int`. No problem. Each compiler knows how big an `int` is. When compilers see the definition for `p`, however, they know they have to allocate enough space for a `Person`, but how are they supposed to know how big a `Person` object is? The only way they can get that information is to consult the class definition, but if it were legal for a class definition to omit the implementation details, how would compilers know how much space to allocate?

In principle, this is no insuperable problem. Languages such as Smalltalk, Eiffel, and Java get around it all the time. The way they do it is by allocating only enough space for a *pointer* to an object when an object is defined. That is, they handle the code above as if it had been written like this:

```
int main()
{
  int x;                    // define an int



  Person *p;                // define a pointer
                            // to a Person
  ...
}
```

It may have occurred to you that this is in fact legal C++, and it turns out that you can play the "hide the object implementation behind a pointer" game yourself.

Here's how you employ the technique to decouple `Person`'s interface from its implementation. First, you put only the following in the header file declaring the `Person` class:

```
// compilers still need to know about these type
// names for the Person constructor
class string;      // again, see Item 49 for information
                   // on why this isn't correct for string
class Date;
class Address;
class Country;



// class PersonImpl will contain the implementation
// details of a Person object; this is just a
// forward declaration of the class name
class PersonImpl;



class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();



  ...                               // copy ctor, operator=



  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;



private:
  PersonImpl *impl;                 // pointer to implementation
};
```

Now clients of `Person` are completely divorced from the details of strings, dates, addresses, countries, and persons. Those classes can be modified at will, but `Person` clients may remain blissfully unaware. More to the point, they may remain blissfully un-recompiled. In addition, because they're unable to see the details of `Person`'s implementation, clients are unlikely to write

code that somehow depends on those details. This is a true separation of interface and implementation.

The key to this separation is replacement of dependencies on class *definitions* with dependencies on class *declarations*. That's all you need to know about minimizing compilation dependencies: make your header files self-sufficient whenever it's practical, and when it's not practical, be dependent on class declarations, not class definitions. Everything else flows from this simple design strategy.

There are three immediate implications:

- **Avoid using objects when object references and pointers will do.** You may define references and pointers to a type with only a *declaration* for the type. Defining *objects* of a type necessitates the presence of the type's definition.
- **Use class declarations instead of class definitions whenever you can.** Note that you *never* need a class definition to declare a function using that class, not even if the function passes or returns the class type by value:

```
class Date;                     // class declaration


Date returnADate();             // fine — no definition
void takeADate(Date d);         // of Date is needed
```

  Of course, pass-by-value is generally a bad idea (see Item 22), but if you find yourself forced to use it for some reason, there's still no justification for introducing unnecessary compilation dependencies.

  If you're surprised that the declarations for `returnADate` and `takeADate` compile without a definition for `Date`, join the club; so was I. It's not as curious as it seems, however, because if anybody *calls* those functions, `Date`'s definition must be visible. Oh, I know what you're thinking: why bother to declare functions that nobody calls? Simple. It's not that *nobody* calls them, it's that *not everybody* calls them. For example, if you have a library containing hundreds of function declarations (possibly spread over several namespaces — see Item 28), it's unlikely that every client calls every function. By moving the onus of providing class definitions (via `#include` directives) from your header file of function *declarations* to clients' files containing function *calls*, you eliminate artificial client dependencies on type definitions they don't really need.
- **Don't `#include` header files in your header files unless your headers won't compile without them.** Instead, manually declare the classes you need, and let clients of your header files `#include` the additional headers necessary to make *their* code compile. A few clients may grumble that this is inconvenient, but rest assured that you are saving them much more pain than you're inflicting. In fact, this technique is so well-regarded, it's enshrined in the standard C++ library (see Item 49); the header `<iosfwd>` contains declarations (and only declarations) for the types in the iostream library.

Classes like `Person` that contain only a pointer to an unspecified implementation are often called *Handle* classes or *Envelope* classes. (In the former case, the classes they point to are called *Body* classes; in latter case, the pointed-to classes are known as *Letter* classes.) Occasionally, you may hear people refer to such classes as *Cheshire Cat* classes, an allusion to the cat in *Alice in Wonderland* that could, when it chose, leave behind only its smile after the rest of it had vanished.

Lest you wonder how Handle classes actually do anything, the answer is simple: they forward all their function calls to the corresponding Body classes, and those classes do the real work. For example, here's how two of `Person`'s member functions would be implemented:

```
#include "Person.h"          // because we're implementing
                             // the Person class, we must
                             // #include its class definition

#include "PersonImpl.h"      // we must also #include
                            // PersonImpl's class definition,
                             // otherwise we couldn't call
                             // its member functions. Note
                             // that PersonImpl has exactly
                             // the same member functions as
                             // Person — their interfaces
                             // are identical

Person::Person(const string& name, const Date& birthday,
               const Address& addr, const Country& country)
{
  impl = new PersonImpl(name, birthday, addr, country);
}

string Person::name() const
{
  return impl->name();
}
```

Note how the `Person` constructor calls the `PersonImpl` constructor (implicitly, by using `new` — see Items 5 and M8) and how `Person::name` calls `PersonImpl::name`. This is important. Making `Person` a handle class doesn't change what `Person` does, it just changes where it does it.

An alternative to the Handle class approach is to make `Person` a special kind of abstract base class called a *Protocol class*. By definition, a Protocol class has no implementation; its *raison d'être* is to specify an interface for derived classes (see Item 36). As a result, it typically has no data members, no constructors, a virtual destructor (see Item 14), and a set of pure virtual functions that specify the interface. A Protocol class for `Person` might look like this:

```
class Person {
public:
```

```
  virtual ~Person();


  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

Clients of this `Person` class must program in terms of `Person` pointers and references, because it's not possible to instantiate classes containing pure virtual functions. (It is, however, possible to instantiate classes *derived* from `Person` — see below.) Like clients of Handle classes, clients of Protocol classes need not recompile unless the Protocol class's interface is modified.

Of course, clients of a Protocol class must have *some* way of creating new objects. They typically do it by calling a function that plays the role of the constructor for the hidden (derived) classes that are actually instantiated. Such functions go by several names (among them *factory functions* and *virtual constructors*), but they all behave the same way: they return pointers to dynamically allocated objects that support the Protocol class's interface (see also Item M25). Such a function might be declared like this,

```
// makePerson is a "virtual constructor" (aka, a "factory
// function") for objects supporting the Person interface
Person*
  makePerson(const string& name,         // return a ptr to
             const Date& birthday,       // a new Person
             const Address& addr,        // initialized with
             const Country& country);    // the given params
```

and used by clients like this:

```
string name;
Date dateOfBirth;
Address address;
Country nation;

...

// create an object supporting the Person interface
Person *pp = makePerson(name, dateOfBirth, address, nation);

...

cout  << pp->name()              // use the object via the
      << " was born on "         // Person interface
      << pp->birthDate()
      << " and now lives at "
      << pp->address();
```

```
...

delete pp;                          // delete the object when
                                    // it's no longer needed
```

Because functions like `makePerson` are closely associated with the Protocol class whose interface is supported by the objects they create, it's good style to declare them `static` inside the Protocol class:

```
class Person {
public:
  ...        // as above



// makePerson is now a member of the class
  static Person * makePerson(const string& name,
                             const Date& birthday,
                             const Address& addr,
                             const Country& country);
};
```

This avoids cluttering the global namespace (or any other namespace) with lots of functions of this nature (see also Item 28).

At some point, of course, concrete classes supporting the Protocol class's interface must be defined and real constructors must be called. That all happens behind the scenes inside the implementation files for the virtual constructors. For example, the Protocol class `Person` might have a concrete derived class `RealPerson` that provides implementations for the virtual functions it inherits:

```
class RealPerson: public Person {
public:
  RealPerson(const string& name, const Date& birthday,
             const Address& addr, const Country& country)
  :  name_(name), birthday_(birthday),
     address_(addr), country_(country)
  {}


  virtual ~RealPerson() {}



  string name() const;          // implementations of
  string birthDate() const;     // these functions are not
  string address() const;       // shown, but they are
```

```
  string nationality() const;   // easy to imagine


private:
  string name_;
  Date birthday_;
  Address address_;
  Country country_;
};
```

Given `RealPerson`, it is truly trivial to write `Person::makePerson`:

```
Person * Person::makePerson(const string& name,
                            const Date& birthday,
                            const Address& addr,
                            const Country& country)
{
  return new RealPerson(name, birthday, addr, country);
}
```

`RealPerson` demonstrates one of the two most common mechanisms for implementing a Protocol class: it inherits its interface specification from the Protocol class (`Person`), then it implements the functions in the interface. A second way to implement a Protocol class involves multiple inheritance, a topic explored in Item 43.

Okay, so Handle classes and Protocol classes decouple interfaces from implementations, thereby reducing compilation dependencies between files. Cynic that you are, I know you're waiting for the fine print. "What does all this hocus-pocus cost me?" you mutter. The answer is the usual one in Computer Science: it costs you some speed at runtime, plus some additional memory per object.

In the case of Handle classes, member functions have to go through the implementation pointer to get to the object's data. That adds one level of indirection per access. And you must add the size of this implementation pointer to the amount of memory required to store each object. Finally, the implementation pointer has to be initialized (in the Handle class's constructors) to point to a dynamically allocated implementation object, so you incur the overhead inherent in dynamic memory allocation (and subsequent deallocation) — see Item 10.

For Protocol classes, every function call is virtual, so you pay the cost of an indirect jump each time you make a function call (see Items 14 and M24). Also, objects derived from the Protocol class must contain a virtual pointer (again, see Items 14 and M24). This pointer may increase the amount of memory needed to store an object, depending on whether the Protocol class is the exclusive source of virtual functions for the object.

Finally, neither Handle classes nor Protocol classes can get much use out of inline functions. All practical uses of inlines require access to implementation details, and that's the very thing that Handle classes and Protocol classes are designed to avoid in the first place.

It would be a serious mistake, however, to dismiss Handle classes and Protocol classes simply because they have a cost associated with them. So do virtual functions, and you wouldn't want to forgo those, would you? (If so, you're reading the wrong book.) Instead, consider using these techniques in an evolutionary manner. Use Handle classes and Protocol classes during development to minimize the impact on clients when implementations change. Replace Handle classes and Protocol classes with concrete classes for production use when it can be shown that the difference in speed and/or size is significant enough to justify the increased coupling between classes. Someday, we may hope, tools will be available to perform this kind of transformation automatically.

A skillful blending of Handle classes, Protocol classes, and concrete classes will allow you to develop software systems that execute efficiently and are easy to evolve, but there is a serious disadvantage: you may have to cut down on the long breaks you've been taking while your programs recompile.

Back to Item 33: Use inlining judiciously.
Continue to Inheritance and Object-Oriented Design

Back to Item 34: Minimize compilation dependencies between files.
Continue to Item 35: Make sure public inheritance models "isa."

Inheritance and Object-Oriented Design

Many people are of the opinion that inheritance is what object-oriented programming is all about. Whether that's so is debatable, but the number of Items in the other sections of this book should convince you that as far as effective C++ programming is concerned, you have a lot more tools at your disposal than simply specifying which classes inherit from which other classes.

Still, designing and implementing class hierarchies is fundamentally different from anything found in the world of C. Certainly it is in the area of inheritance and object-oriented design that you are most likely to radically rethink your approach to the construction of software systems. Furthermore, C++ provides a bewildering assortment of object-oriented building blocks, including public, protected, and private base classes; virtual and nonvirtual base classes; and virtual and nonvirtual member functions. Each of these features interacts not only with one another, but also with the other components of the language. As a result, trying to understand what each feature means, when it should be used, and how it is best combined with the non-object-oriented aspects of C++ can be a daunting endeavor.

Further complicating the matter is the fact that different features of the language appear to do more or less the same thing. Examples:

- You need a collection of classes with many shared characteristics. Should you use inheritance and have all the classes derived from a common base class, or should you use templates and have them all generated from a common code skeleton?
- Class A is to be implemented in terms of class B. Should A have a data member of type B, or should A privately inherit from B?
- You need to design a type-safe homogeneous container class, one not present in the standard library. (See Item 49 for a list of containers the library *does* provide.) Should you use templates, or would it be better to build type-safe interfaces around a class that is itself implemented using generic (`void*`) pointers?

In the Items in this section, I offer guidance on how to answer questions such as these. However, I cannot hope to address every aspect of object-oriented design. Instead, I concentrate on explaining what the different features in C++ really *mean*, on what you are really *saying* when you use a particular feature. For example, public inheritance means "isa" (see Item 35), and if you try to make it mean anything else, you will run into trouble. Similarly, a virtual function means "interface must be inherited," while a nonvirtual function means "both interface *and* implementation must be inherited." Failing to distinguish between these meanings has caused many a C++ programmer untold grief.

If you understand the meanings of C++'s varied features, you'll find that your outlook on object-oriented design shifts. Instead of it being an exercise in differentiating between language constructs, it will properly become a matter of figuring out what it is you want to say about your software system. Once you know what you want to say, you'll be able to translate that into the appropriate C++ features without too much difficulty.

The importance of saying what you mean and understanding what you're saying cannot be overestimated. The items that follow provide a detailed examination of how to do this effectively. Item 44 summarizes the correspondence between C++'s object-oriented constructs and what they mean. It serves as a nice capstone for this section, as well as a concise reference for future consultation.

Back to Item 34: Minimize compilation dependencies between files.
Continue to Item 35: Make sure public inheritance models "isa."

Back to Inheritance and Object-Oriented Design
Continue to Item 36: Differentiate between inheritance of interface and inheritance of implementation.

Item 35: Make sure public inheritance models "isa."

In his book, *Some Must Watch While Some Must Sleep* (W. H. Freeman and Company, 1974), William Dement relates the story of his attempt to fix in the minds of his students the most important lessons of his course. It is claimed, he told his class, that the average British schoolchild remembers little more history than that the Battle of Hastings was in 1066. If a child remembers little else, Dement emphasized, he or she remembers the date 1066. For the students in *his* course, Dement went on, there were only a few central messages, including, interestingly enough, the fact that sleeping pills cause insomnia. He implored his students to remember these few critical facts even if they forgot everything else discussed in the course, and he returned to these fundamental precepts repeatedly during the term.

At the end of the course, the last question on the final exam was, "Write one thing from the course that you will surely remember for the rest of your life." When Dement graded the exams, he was stunned. Nearly everyone had written "1066."

It is thus with great trepidation that I proclaim to you now that the single most important rule in object-oriented programming with C++ is this: public inheritance means "isa." Commit this rule to memory.

If you write that class D ("Derived") publicly inherits from class B ("Base"), you are telling C++ compilers (as well as human readers of your code) that every object of type D is also an object of type B, but *not vice versa*. You are saying that B represents a more general concept than D, that D represents a more specialized concept than B. You are asserting that anywhere an object of type B can be used, an object of type D can be used just as well, because every object of type D *is* an object of type B. On the other hand, if you need an object of type D, an object of type B will not do: every D isa B, but not vice versa.

C++ enforces this interpretation of public inheritance. Consider this example:

```
class Person { ... };


class Student: public Person { ... };
```

We know from everyday experience that every student is a person, but not every person is a student. That is exactly what this hierarchy asserts. We expect that anything that is true of a person — for example, that he or she has a date of birth — is also true of a student, but we do not expect that

everything that is true of a student — that he or she is enrolled in a particular school, for instance — is true of people in general. The notion of a person is more general than is that of a student; a student is a specialized type of person.

Within the realm of C++, any function that expects an argument of type `Person` (or pointer-to-`Person` or reference-to-`Person`) will instead take a `Student` object (or pointer-to-`Student` or reference-to-`Student`):

```
void dance(const Person& p);        // anyone can dance


void study(const Student& s);       // only students study


Person p;                           // p is a Person
Student s;                          // s is a Student


dance(p);                           // fine, p is a Person


dance(s);                           // fine, s is a Student,
                                    // and a Student isa Person


study(s);                           // fine


study(p);                           // error! p isn't a Student
```

This is true only for *public* inheritance. C++ will behave as I've described only if `Student` is publicly derived from `Person`. Private inheritance means something entirely different (see Item 42), and no one seems to know what protected inheritance is supposed to mean. Furthermore, the fact that a `Student` isa `Person` does *not* mean that an *array* of `Student` isa *array* of `Person`. For more information on that topic, see Item M3.

The equivalence of public inheritance and isa sounds simple, but in practice, things aren't always so

straightforward. Sometimes your intuition can mislead you. For example, it is a fact that a penguin is a bird, and it is a fact that birds can fly. If we naively try to express this in C++, our effort yields:

```
class Bird {
public:
  virtual void fly();              // birds can fly


  ...


};



class Penguin:public Bird {      // penguins are birds


  ...


};
```

Suddenly we are in trouble, because this hierarchy says that penguins can fly, which we know is not true. What happened?

In this case, we are the victims of an imprecise language (English). When we say that birds can fly, we don't really mean that *all* birds can fly, only that, in general, birds have the ability to fly. If we were more precise, we'd recognize that there are in fact several types of non-flying birds, and we would come up with the following hierarchy, which models reality much better:

```
class Bird {
  ...                    // no fly function is
};                       // declared


class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};


class NonFlyingBird: public Bird {
```

```
  ...                   // no fly function is
                        // declared
};


class Penguin: public NonFlyingBird {



  ...                   // no fly function is
                        // declared
};
```

This hierarchy is much more faithful to what we really know than was the original design.

Even now we're not entirely finished with these fowl matters, because for some software systems, it may be entirely appropriate to say that a penguin is a bird. In particular, if your application has much to do with beaks and wings and nothing to do with flying, the original hierarchy might work out just fine. Irritating though this may seem, it's a simple reflection of the fact that there is no one ideal design for all software. The best design depends on what the system is expected to do, both now and in the future (see Item M32). If your application has no knowledge of flying and isn't expected to ever have any, making `Penguin` a derived class of `Bird` may be a perfectly valid design decision. In fact, it may be preferable to a decision that makes a distinction between flying and non-flying birds, because such a distinction would be absent from the world you are trying to model. Adding superfluous classes to a hierarchy can be just as bad a design decision as having the wrong inheritance relationships between classes.

There is another school of thought on how to handle what I call the "All birds can fly, penguins are birds, penguins can't fly, uh oh" problem. That is to redefine the `fly` function for penguins so that it generates a runtime error:

```
void error(const string& msg);      // defined elsewhere


class Penguin: public Bird {
public:
  virtual void fly() { error("Penguins can't fly!"); }


  ...
```

```
};
```

Interpreted languages such as Smalltalk tend to adopt this approach, but it's important to recognize that this says something entirely different from what you might think. This does *not* say, "Penguins can't fly." This says, "Penguins can fly, but it's an error for them to try to do so."

How can you tell the difference? From the time at which the error is detected. The injunction, "Penguins can't fly," can be enforced by compilers, but violations of the statement, "It's an error for penguins to try to fly," can be detected only at runtime.

To express the constraint, "Penguins can't fly," you make sure that no such function is defined for `Penguin` objects:

```
class Bird {


  ...                          // no fly function is
                               // declared
};


class NonFlyingBird: public Bird {


  ...                          // no fly function is
                               // declared
};


class Penguin: public NonFlyingBird {


  ...                          // no fly function is
                               // declared
};
```

If you try to make a penguin fly, compilers will reprimand you for your transgression:

```
Penguin p;
```

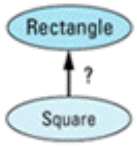
```
p.fly();                        // error!
```

This is very different from the behavior you get if you use the Smalltalk approach. With that methodology, compilers won't say a word.

The C++ philosophy is fundamentally different from the Smalltalk philosophy, so you're better off doing things the C++ way as long as you're programming in C++. In addition, there are certain technical advantages to detecting errors during compilation instead of at runtime — see Item 46.

Perhaps you'll concede that your ornithological intuition may be lacking, but you can rely on your mastery of elementary geometry, right? I mean, how complicated can rectangles and squares be?

Well, answer this simple question: should class `Square` publicly inherit from class `Rectangle`?



"Duh!" you say, "Of course it should! Everybody knows that a square is a rectangle, but generally not vice versa." True enough, at least in high school. But I don't think we're in high school anymore.

Consider this code:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);



  virtual int height() const;          // return current
  virtual int width() const;           // values


  ...


};
```

```
void makeBigger(Rectangle& r)          // function to
{                                      // increase r's area
  int oldHeight = r.height();



  r.setWidth(r.width() + 10);          // add 10 to r's width



  assert(r.height() == oldHeight);     // assert that r's
}                                      // height is unchanged
```

Clearly, the assertion should never fail. `makeBigger` only changes `r`'s width. Its height is never modified.

Now consider this code, which uses public inheritance to allow squares to be treated like rectangles:

```
class Square: public Rectangle { ... };


Square s;


...



assert(s.width() == s.height());      // this must be true
                                      // for all squares



makeBigger(s);                        // by inheritance, s
                                      // isa Rectangle, so
                                      // we can increase its
                                      // area



assert(s.width() == s.height());      // this must still be
                                      // true for all squares
```

It's just as clear here as it was above that this last assertion should also never fail. By definition, the width of a square is the same as its height.

But now we have a problem. How can we reconcile the following assertions?

- Before calling `makeBigger`, `s`'s height is the same as its width;
- Inside `makeBigger`, `s`'s width is changed, but its height is not;
- After returning from `makeBigger`, `s`'s height is again the same as its width. (Note that `s` is passed to `makeBigger` by reference, so `makeBigger` modifies `s` itself, not a copy of `s`.)

Well?

Welcome to the wonderful world of public inheritance, where the instincts you've developed in other fields of study — including mathematics — may not serve you as well as you expect. The fundamental difficulty in this case is that something applicable to a rectangle (its width may be modified independently of its height) is not applicable to a square (its width and height are constrained to be the same). But public inheritance asserts that everything applicable to base class objects — *everything!* — is also applicable to derived class objects. In the case of rectangles and squares (and a similar example involving sets and lists in Item 40), that assertion fails to hold, so using public inheritance to model their relationship is just plain wrong. Compilers will let you do it, of course, but as we've just seen, that's no guarantee the code will behave properly. As every programmer must learn (some more often than others), just because the code compiles doesn't mean it will work.

Now, don't fret that the software intuition you've developed over the years will fail you as you approach object-oriented design. That knowledge is still valuable, but now that you've added inheritance to your arsenal of design alternatives, you'll have to augment your intuition with new insights to guide you in inheritance's proper application. In time, the notion of having `Penguin` inherit from `Bird` or `Square` inherit from `Rectangle` will give you the same funny feeling you probably get now when somebody shows you a function several pages long. It's *possible* that it's the right way to approach things, it's just not very likely.

Of course, the isa relationship is not the only one that can exist between classes. Two other common inter-class relationships are "has-a" and "is-implemented-in-terms-of." These relationships are considered in Items 40 and 42. It's not uncommon for C++ designs to go awry because one of these other important relationships was incorrectly modeled as isa, so you should make sure that you understand the differences between these relationships and that you know how they are best modeled in C++.

Back to Inheritance and Object-Oriented Design
Continue to Item 36: Differentiate between inheritance of interface and inheritance of implementation.

Back to Item 35: Make sure public inheritance models "isa."
Continue to Item 37: Never redefine an inherited nonvirtual function.

Item 36: Differentiate between inheritance of interface and inheritance of implementation.

The seemingly straightforward notion of (public) inheritance turns out, upon closer examination, to be composed of two separable parts: inheritance of function interfaces and inheritance of function implementations. The difference between these two kinds of inheritance corresponds exactly to the difference between function declarations and function definitions discussed in the Introduction to this book.

As a class designer, you sometimes want derived classes to inherit only the interface (declaration) of a member function; sometimes you want derived classes to inherit both the interface and the implementation for a function, but you want to allow them to override the implementation you provide; and sometimes you want them to inherit both interface and implementation without allowing them to override anything.

To get a better feel for the differences among these options, consider a class hierarchy for representing geometric shapes in a graphics application:

```
class Shape {
public:
  virtual void draw() const = 0;


  virtual void error(const string& msg);


  int objectID() const;


  ...


};


class Rectangle: public Shape { ... };


class Ellipse: public Shape { ... };
```

`Shape` is an abstract class; its pure virtual function `draw` marks it as such. As a result, clients cannot create instances of the `Shape` class, only of the classes derived from it. Nonetheless, `Shape` exerts a strong influence on all classes that (publicly) inherit from it, because

- Member function *interfaces are always inherited*. As explained in Item 35, public inheritance means isa, so anything that is true of a base class must also be true of its derived classes. Hence, if a function applies to a class, it must also apply to its subclasses.

Three functions are declared in the `Shape` class. The first, `draw`, draws the current object on an implicit display. The second, `error`, is called by member functions if they need to report an error. The third, `objectID`, returns a unique integer identifier for the current object; Item 17 gives an example of how such a function might be used. Each function is declared in a different way: `draw` is a pure virtual function; `error` is a simple (impure?) virtual function; and `objectID` is a nonvirtual function. What are the implications of these different declarations?

Consider first the pure virtual function `draw`. The two most salient features of pure virtual functions are that they *must* be redeclared by any concrete class that inherits them, and they typically have no definition in abstract classes. Put these two traits together, and you realize that

- The purpose of declaring a pure virtual function is to have derived classes inherit a function *interface only*.

This makes perfect sense for the `Shape::draw` function, because it is a reasonable demand that all `Shape` objects must be `draw`able, but the `Shape` class can provide no reasonable default implementation for that function. The algorithm for drawing an ellipse is very different from the algorithm for drawing a rectangle, for example. A good way to interpret the declaration of `Shape::draw` is as saying to designers of subclasses, "You must provide a `draw` function, but I have no idea how you're going to implement it."

Incidentally, it *is* possible to provide a definition for a pure virtual function. That is, you could provide an implementation for `Shape::draw`, and C++ wouldn't complain, but the only way to call it would be to fully specify the call with the class name:

```
Shape *ps = new Shape;          // error! Shape is abstract


Shape *ps1 = new Rectangle;     // fine
ps1->draw();                    // calls Rectangle::draw
```

```
Shape *ps2 = new Ellipse;        // fine
ps2->draw();                     // calls Ellipse::draw



ps1->Shape::draw();              // calls Shape::draw



ps2->Shape::draw();              // calls Shape::draw
```

Aside from helping impress fellow programmers at cocktail parties, knowledge of this feature is generally of limited utility. As you'll see below, however, it can be employed as a mechanism for providing a safer-than-usual default implementation for simple (impure) virtual functions.

Sometimes it's useful to declare a class containing *nothing* but pure virtual functions. Such a *Protocol class* can provide only function interfaces for derived classes, never implementations. Protocol classes are described in Item 34 and are mentioned again in Item 43.

The story behind simple virtual functions is a bit different from that behind pure virtuals. As usual, derived classes inherit the interface of the function, but simple virtual functions traditionally provide an implementation that derived classes may or may not choose to override. If you think about this for a minute, you'll realize that

- The purpose of declaring a simple virtual function is to have derived classes inherit a function *interface as well as a default implementation.*

In the case of `Shape::error`, the interface says that every class must support a function to be called when an error is encountered, but each class is free to handle errors in whatever way it sees fit. If a class doesn't want to do anything special, it can just fall back on the default error-handling provided in the `Shape` class. That is, the declaration of `Shape::error` says to designers of subclasses, "You've got to support an `error` function, but if you don't want to write your own, you can fall back on the default version in the `Shape` class."

It turns out that it can be dangerous to allow simple virtual functions to specify both a function declaration and a default implementation. To see why, consider a hierarchy of airplanes for XYZ Airlines. XYZ has only two kinds of planes, the Model A and the Model B, and both are flown in exactly the same way. Hence, XYZ designs the following hierarchy:

```
class Airport { ... };      // represents airports


class Airplane {
public:
  virtual void fly(const Airport& destination);


  ...


};


void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to
  the given destination
}


class ModelA: public Airplane { ... };


class ModelB: public Airplane { ... };
```

To express that all planes have to support a `fly` function, and in recognition of the fact that different models of plane could, in principle, require different implementations for `fly`, `Airplane::fly` is declared virtual. However, in order to avoid writing identical code in the `ModelA` and `ModelB` classes, the default flying behavior is provided as the body of `Airplane::fly`, which both `ModelA` and `ModelB` inherit.

This is a classic object-oriented design. Two classes share a common feature (the way they implement `fly`), so the common feature is moved into a base class, and the feature is inherited by the two classes. This design makes common features explicit, avoids code duplication, facilitates future enhancements, and eases long-term maintenance — all the things for which object-oriented technology is so highly touted. XYZ Airlines should be proud.

Now suppose that XYZ, its fortunes on the rise, decides to acquire a new type of airplane, the Model C. The Model C differs from the Model A and the Model B. In particular, it is flown differently.

XYZ's programmers add the class for Model C to the hierarchy, but in their haste to get the new model into service, they forget to redefine the `fly` function:

```
class ModelC: public Airplane {


  ...                          // no fly function is
                               // declared
};
```

In their code, then, they have something akin to the following:

```
Airport JFK(...);              // JFK is an airport in
                               // New York City


Airplane *pa = new ModelC;


...



pa->fly(JFK);                  // calls Airplane::fly!
```

This is a disaster: an attempt is being made to fly a `ModelC` object as if it were a `ModelA` or a `ModelB`. That's not the kind of behavior that inspires confidence in the traveling public.

The problem here is not that `Airplane::fly` has default behavior, but that `ModelC` was allowed to inherit that behavior without explicitly saying that it wanted to. Fortunately, it's easy to offer default behavior to subclasses, but not give it to them unless they ask for it. The trick is to sever the connection between the *interface* of the virtual function and its default *implementation*. Here's one way to do it:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;


  ...



protected:
  void defaultFly(const Airport& destination);
};
```

```
void Airplane::defaultFly(const Airport& destination)
{
  default code for flying an airplane to
  the given destination
}
```

Notice how `Airplane::fly` has been turned into a pure virtual function. That provides the interface for flying. The default implementation is also present in the `Airplane` class, but now it's in the form of an independent function, `defaultFly`. Classes like `ModelA` and `ModelB` that want to use the default behavior simply make an inline call to `defaultFly` inside their body of `fly` (but see Item 33 for information on the interaction of inlining and virtual functions):

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }


  ...


};


class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }


  ...


};
```

For the `ModelC` class, there is no possibility of accidentally inheriting the incorrect implementation of `fly`, because the pure virtual in `Airplane` forces `ModelC` to provide its own version of `fly`.

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
```

```
};


void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

This scheme isn't foolproof (programmers can still copy-and-paste themselves into trouble), but it's more reliable than the original design. As for `Airplane::defaultFly`, it's protected because it's truly an implementation detail of `Airplane` and its derived classes. Clients using airplanes should care only that they can be flown, not how the flying is implemented.

It's also important that `Airplane::defaultFly` is a *nonvirtual* function. This is because no subclass should redefine this function, a truth to which Item 37 is devoted. If `defaultFly` were virtual, you'd have a circular problem: what if some subclass forgets to redefine `defaultFly` when it's supposed to?

Some people object to the idea of having separate functions for providing interface and default implementation, such as `fly` and `defaultFly` above. For one thing, they note, it pollutes the class namespace with a proliferation of closely-related function names. Yet they still agree that interface and default implementation should be separated. How do they resolve this seeming contradiction? By taking advantage of the fact that pure virtual functions must be redeclared in subclasses, but they may also have implementations of their own. Here's how the `Airplane` hierarchy could take advantage of the ability to define a pure virtual function:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;


  ...


};


void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to
  the given destination
}
```

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

This is almost exactly the same design as before, except that the body of the pure virtual function `Airplane::fly` takes the place of the independent function `Airplane::defaultFly`. In essence, `fly` has been broken into its two fundamental components. Its declaration specifies its interface (which derived classes *must* use), while its definition specifies its default behavior (which derived classes *may* use, but only if they explicitly request it). In merging `fly` and `defaultFly`, however,

you've lost the ability to give the two functions different protection levels: the code that used to be `protected` (by being in `defaultFly`) is now `public` (because it's in `fly`).

Finally, we come to `Shape`'s nonvirtual function, `objectID`. When a member function is nonvirtual, it's not supposed to behave differently in derived classes. In fact, a nonvirtual member function specifies an *invariant over specialization*, because it identifies behavior that is not supposed to change, no matter how specialized a derived class becomes. As such,

- The purpose of declaring a nonvirtual function is to have derived classes inherit a function *interface as well as a mandatory implementation.*

You can think of the declaration for `Shape::objectID` as saying, "Every `Shape` object has a function that yields an object identifier, and that object identifier is always computed in the same way. That way is determined by the definition of `Shape::objectID`, and no derived class should try to change how it's done." Because a nonvirtual function identifies an *invariant* over specialization, it should never be redefined in a subclass, a point that is discussed in detail in Item 37.

The differences in declarations for pure virtual, simple virtual, and nonvirtual functions allow you to specify with precision what you want derived classes to inherit: interface only, interface and a default implementation, or interface and a mandatory implementation, respectively. Because these different types of declarations mean fundamentally different things, you must choose carefully among them when you declare your member functions. If you do, you should avoid the two most common mistakes made by inexperienced class designers.

The first mistake is to declare all functions nonvirtual. That leaves no room for specialization in derived classes; nonvirtual destructors are particularly problematic (see Item 14). Of course, it's perfectly reasonable to design a class that is not intended to be used as a base class. Item M34 gives an example of a case where you might want to. In that case, a set of exclusively nonvirtual member functions is appropriate. Too often, however, such classes are declared either out of ignorance of the differences between virtual and nonvirtual functions or as a result of an unsubstantiated concern over the performance cost of virtual functions (see Item M24). The fact of the matter is that almost any class that's to be used as a base class will have virtual functions (again, see Item 14).

If you're concerned about the cost of virtual functions, allow me to bring up the rule of 80-20 (see Item M16), which states that in a typical program, 80 percent of the runtime will be spent executing just 20 percent of the code. This rule is important, because it means that, on average, 80 percent of your function calls can be virtual without having the slightest detectable impact on your program's overall performance. Before you go gray worrying about whether you can afford the cost of a virtual function, then, take the simple precaution of making sure that you're focusing on the 20 percent of your program where the decision might really make a difference.

The other common problem is to declare *all* member functions virtual. Sometimes this is the right thing to do — witness Protocol classes (see Item 34), for example. However, it can also be a sign of a class designer who lacks the backbone to take a firm stand. Some functions should *not* be redefinable in derived classes, and whenever that's the case, you've got to say so by making those functions nonvirtual. It serves no one to pretend that your class can be all things to all people if they'll just take the time to redefine all your functions. Remember that if you have a base class `B`, a derived class `D`, and a member function `mf`, then each of the following calls to `mf` *must* work properly:

```
D *pd = new D;
B *pb = pd;



pb->mf();                   // call mf through a
                            // pointer-to-base



pd->mf();                   // call mf through a
                            // pointer-to-derived
```

Sometimes, you must make `mf` a nonvirtual function to ensure that everything behaves the way it's supposed to (see Item 37). If you have an invariant over specialization, don't be afraid to say so!

Back to Item 35: Make sure public inheritance models "isa."
Continue to Item 37: Never redefine an inherited nonvirtual function.

Back to Item 36: Differentiate between inheritance of interface and inheritance of implementation.
Continue to Item 38: Never redefine an inherited default parameter value.

Item 37: Never redefine an inherited nonvirtual function.

There are two ways of looking at this issue: the theoretical way and the pragmatic way. Let's start with the pragmatic way. After all, theoreticians are used to being patient.

Suppose I tell you that a class `D` is publicly derived from a class `B` and that there is a public member function `mf` defined in class `B`. The parameters and return type of `mf` are unimportant, so let's just assume they're both `void`. In other words, I say this:

```
class B {
public:
  void mf();
  ...
};


class D: public B { ... };
```

Even without knowing anything about `B`, `D`, or `mf`, given an object `x` of type `D`,

```
D x;                            // x is an object of type D
```

you would probably be quite surprised if this,

```
B *pB = &x;                     // get pointer to x


pB->mf();                       // call mf through pointer
```

behaved differently from this:

```
D *pD = &x;                     // get pointer to x
```

```
pD->mf();                    // call mf through pointer
```

That's because in both cases you're invoking the member function `mf` on the object `x`. Because it's the same function and the same object in both cases, it should behave the same way, right?

Right, it should. But it might not. In particular, it won't if `mf` is nonvirtual and `D` has defined its own version of `mf`:

```
class D: public B {
public:
  void mf();                 // hides B::mf; see Item 50


  ...


};



pB->mf();                    // calls B::mf



pD->mf();                    // calls D::mf
```

The reason for this two-faced behavior is that *nonvirtual* functions like `B::mf` and `D::mf` are statically bound (see Item 38). That means that because `pB` is declared to be of type pointer-to-`B`, nonvirtual functions invoked through `pB` will *always* be those defined for class `B`, even if `pB` points to an object of a class derived from `B`, as it does in this example.

Virtual functions, on the other hand, are dynamically bound (again, see Item 38), so they don't suffer from this problem. If `mf` were a virtual function, a call to `mf` through either `pB` or `pD` would result in an invocation of `D::mf`, because what `pB` and `pD` *really* point to is an object of type `D`.

The bottom line, then, is that if you are writing class `D` and you redefine a nonvirtual function `mf` that you inherit from class `B`, `D` objects will likely exhibit schizophrenic behavior. In particular, any given `D` object may act like either a `B` or a `D` when `mf` is called, and the determining factor will have nothing to do with the object itself, but with the declared type of the pointer that points to it. References exhibit the same baffling behavior as do pointers.

So much for the pragmatic argument. What you want now, I know, is some kind of theoretical justification for not redefining inherited nonvirtual functions. I am pleased to oblige.

Item 35 explains that public inheritance means isa, and Item 36 describes why declaring a nonvirtual function in a class establishes an invariant over specialization for that class. If you apply these observations to the classes `B` and `D` and to the nonvirtual member function `B::mf`, then

- Everything that is applicable to `B` objects is also applicable to `D` objects, because every `D` object isa `B` object;
- Subclasses of `B` must inherit both the interface *and* the implementation of `mf`, because `mf` is nonvirtual in `B`.

Now, if `D` redefines `mf`, there is a contradiction in your design. If `D` *really* needs to implement `mf` differently from `B`, and if every `B` object — no matter how specialized — *really* has to use the `B` implementation for `mf`, then it's simply not true that every `D` isa `B`. In that case, `D` shouldn't publicly inherit from `B`. On the other hand, if `D` *really* has to publicly inherit from `B`, and if `D` *really* needs to implement `mf` differently from `B`, then it's just not true that `mf` reflects an invariant over specialization for `B`. In that case, `mf` should be virtual. Finally, if every `D` *really* isa `B`, and if `mf` really corresponds to an invariant over specialization for `B`, then `D` can't honestly need to redefine `mf`, and it shouldn't try to do so.

Regardless of which argument applies, something has to give, and under no conditions is it the prohibition on redefining an inherited nonvirtual function.

Back to Item 36: Differentiate between inheritance of interface and inheritance of implementation.
Continue to Item 38: Never redefine an inherited default parameter value.

Back to Item 37: Never redefine an inherited nonvirtual function.
Continue to Item 39: Avoid casts down the inheritance hierarchy.

Item 38: Never redefine an inherited default parameter value.

Let's simplify this discussion right from the start. A default parameter can exist only as part of a function, and you can inherit only two kinds of functions: virtual and nonvirtual. Therefore, the only way to redefine a default parameter value is to redefine an inherited function. However, it's always a mistake to redefine an inherited nonvirtual function (see Item 37), so we can safely limit our discussion here to the situation in which you inherit a *virtual* function with a default parameter value.

That being the case, the justification for this Item becomes quite straightforward: virtual functions are dynamically bound, but default parameter values are statically bound.

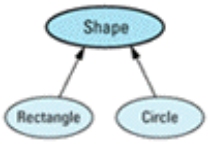
What's that? You say you're not up on the latest object-oriented lingo, or perhaps the difference between static and dynamic binding has slipped your already overburdened mind? Let's review, then.

An object's *static type* is the type you declare it to have in the program text. Consider this class hierarchy:

```
enum ShapeColor { RED, GREEN, BLUE };


// a class for geometric shapes
class Shape {
public:
  // all shapes must offer a function to draw themselves
  virtual void draw(ShapeColor color = RED) const = 0;


  ...


};


class Rectangle: public Shape {
public:
  // notice the different default parameter value - bad!
  virtual void draw(ShapeColor color = GREEN) const;
```

```
  ...

};

class Circle: public Shape {
public:
  virtual void draw(ShapeColor color) const;

  ...

};
```

Graphically, it looks like this:



Now consider these pointers:

```
Shape *ps;                    // static type = Shape*

Shape *pc = new Circle;       // static type = Shape*

Shape *pr = new Rectangle;    // static type = Shape*
```

In this example, `ps`, `pc`, and `pr` are all declared to be of type pointer-to-`Shape`, so they all have that as their static type. Notice that it makes absolutely no difference what they're *really* pointing to — their static type is `Shape*` regardless.

An object's *dynamic type* is determined by the type of the object to which it currently refers. That is, its dynamic type indicates how it will behave. In the example above, `pc`'s dynamic type is `Circle*`, and `pr`'s dynamic type is `Rectangle*`. As for `ps`, it doesn't really have a dynamic type, because it doesn't refer to any object (yet).

Dynamic types, as their name suggests, can change as a program runs, typically through assignments:

```
ps = pc;                        // ps's dynamic type is
                                // now Circle*


ps = pr;                        // ps's dynamic type is
                                // now Rectangle*
```

Virtual functions are *dynamically bound*, meaning that the particular function called is determined by the dynamic type of the object through which it's invoked:

```
pc->draw(RED);                  // calls Circle::draw(RED)


pr->draw(RED);                  // calls Rectangle::draw(RED)
```

This is all old hat, I know; you surely understand virtual functions. (If you'd like to understand how they're implemented, turn to Item M24.) The twist comes in when you consider virtual functions with default parameter values, because, as I said above, virtual functions are dynamically bound, but default parameters are statically bound. That means that you may end up invoking a virtual function defined in a *derived class* but using a default parameter value from a *base class*:

```
pr->draw();                     // calls Rectangle::draw(RED)!
```

In this case, `pr`'s dynamic type is `Rectangle*`, so the `Rectangle` virtual function is called, just as you would expect. In `Rectangle::draw`, the default parameter value is `GREEN`. Because `pr`'s static type is `Shape*`, however, the default parameter value for this function call is taken from the `Shape` class, not the `Rectangle` class! The result is a call consisting of a strange and almost certainly unanticipated combination of the declarations for `draw` in both the `Shape` and `Rectangle` classes. Trust me when I tell you that you don't want your software to behave this way, or at least believe me when I tell you that your *clients* won't want your software to behave this way.

Needless to say, the fact that `ps`, `pc`, and `pr` are pointers is of no consequence in this matter. Were they references, the problem would persist. The only important things are that `draw` is a virtual function, and one of its default parameter values is redefined in a subclass.

Why does C++ insist on acting in this perverse manner? The answer has to do with runtime efficiency. If default parameter values were dynamically bound, compilers would have to come up with a way of determining the appropriate default value(s) for parameters of virtual functions at runtime, which would be slower and more complicated than the current mechanism of determining them during compilation. The decision was made to err on the side of speed and simplicity of implementation, and the result is that you now enjoy execution behavior that is efficient, but, if you fail to heed the advice of this Item, confusing.

Back to Item 37: Never redefine an inherited nonvirtual function.
Continue to Item 39: Avoid casts down the inheritance hierarchy.

Back to Item 38: Never redefine an inherited default parameter value.
Continue to Item 40: Model "has-a" or "is-implemented-in-terms-of" through layering.

Item 39: Avoid casts down the inheritance hierarchy.

In these tumultuous economic times, it's a good idea to keep an eye on our financial institutions, so consider a Protocol class (see Item 34) for bank accounts:

```
class Person { ... };


class BankAccount {
public:
  BankAccount(const Person *primaryOwner,
              const Person *jointOwner);
  virtual ~BankAccount();


  virtual void makeDeposit(double amount) = 0;
  virtual void makeWithdrawal(double amount) = 0;


  virtual double balance() const = 0;


  ...


};
```

Many banks now offer a bewildering array of account types, but to keep things simple, let's assume there is only one type of bank account, namely, a savings account:

```
class SavingsAccount: public BankAccount {
public:
  SavingsAccount(const Person *primaryOwner,
                 const Person *jointOwner);
  ~SavingsAccount();


  void creditInterest();               // add interest to account
```

```
  ...

};
```

This isn't much of a savings account, but then again, what is these days? At any rate, it's enough for our purposes.

A bank is likely to keep a list of all its accounts, perhaps implemented via the `list` class template from the standard library (see Item 49). Suppose this list is imaginatively named `allAccounts`:

```
list<BankAccount*> allAccounts;          // all accounts at the
                                         // bank
```

Like all standard containers, `lists` store *copies* of the things placed into them, so to avoid storing multiple copies of each `BankAccount`, the bank has decided to have `allAccounts` hold *pointers* to `BankAccounts` instead of `BankAccounts` themselves.

Now imagine you're supposed to write the code to iterate over all the accounts, crediting the interest due each one. You might try this,

```
// a loop that won't compile (see below if you've never
// seen code using "iterators" before)
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  (*p)->creditInterest();      // error!


}
```

but your compilers would quickly bring you to your senses: `allAccounts` contains pointers to `BankAccount` objects, not to `SavingsAccount` objects, so each time around the loop, `p` points to a `BankAccount`. That makes the call to `creditInterest` invalid, because `creditInterest` is declared only for `SavingsAccount` objects, not `BankAccounts`.

If "`list<BankAccount*>::iterator p = allAccounts.begin()`" looks to you more like transmission line noise than C++, you've apparently never had the pleasure of meeting the container

class templates in the standard library. This part of the library is usually known as the Standard Template Library (the "STL"), and you can get an overview of it in Items 49 and M35. For the time being, all you need to know is that the variable `p` acts like a pointer that loops through the elements of `allAccounts` from beginning to end. That is, `p` acts as if its type were `BankAccount**` and the list elements were stored in an array.

It's frustrating that the loop above won't compile. Sure, `allAccounts` is defined as holding `BankAccount*`s, but you *know* that it actually holds `SavingsAccount*`s in the loop above, because `SavingsAccount` is the only class that can be instantiated. Stupid compilers! You decide to tell them what you know to be obvious and what they are too dense to figure out on their own: `allAccounts` really contains `SavingsAccount*`s:

```
// a loop that will compile, but that is nonetheless evil
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  static_cast<SavingsAccount*>(*p)->creditInterest();


}
```

All your problems are solved! Solved clearly, solved elegantly, solved concisely, all by the simple use of a cast. You know what type of pointer `allAccounts` really holds, your dopey compilers don't, so you use a cast to tell them. What could be more logical?

There is a biblical analogy I'd like to draw here. Casts are to C++ programmers what the apple was to Eve.

This kind of cast — from a base class pointer to a derived class pointer — is called a *downcast*, because you're casting down the inheritance hierarchy. In the example you just looked at, downcasting happens to work, but it leads to a maintenance nightmare, as you will soon see.

But back to the bank. Buoyed by the success of its savings accounts, let's suppose the bank decides to offer checking accounts, too. Furthermore, assume that checking accounts also bear interest, just like savings accounts:

```
class CheckingAccount: public BankAccount {
public:
  void creditInterest();    // add interest to account
```

```
  ...


};
```

Needless to say, `allAccounts` will now be a list containing pointers to both savings and checking accounts. Suddenly, the interest-crediting loop you wrote above is in serious trouble.

Your first problem is that it will continue to compile without your changing it to reflect the existence of `CheckingAccount` objects. This is because compilers will foolishly believe you when you tell them (through the `static_cast`) that `*p` really points to a `SavingsAccount*`. After all, you're the boss. That's Maintenance Nightmare Number One. Maintenance Nightmare Number Two is what you're tempted to do to fix the problem, which is typically to write code like this:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  if (*p points to a SavingsAccount)
    static_cast<SavingsAccount*>(*p)->creditInterest();
  else
    static_cast<CheckingAccount*>(*p)->creditInterest();


}
```

Anytime you find yourself writing code of the form, "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself. That isn't The C++ Way. Yes, it's a reasonable strategy in C, in Pascal, even in Smalltalk, but not in C++. In C++, you use virtual functions.

Remember that with a virtual function, *compilers* are responsible for making sure that the right function is called, depending on the type of the object being used. Don't litter your code with conditionals or switch statements; let your compilers do the work for you. Here's how:

```
class BankAccount { ... };      // as above


// new class representing accounts that bear interest
class InterestBearingAccount: public BankAccount {
public:
  virtual void creditInterest() = 0;
```

```
  ...

};

class SavingsAccount: public InterestBearingAccount {

  ...                           // as above

};

class CheckingAccount: public InterestBearingAccount {

  ...                           // as above

};
```

Graphically, it looks like this:

Because both savings and checking accounts earn interest, you'd naturally like to move that common behavior up into a common base class. However, under the assumption that not all

accounts in the bank will necessarily bear interest (certainly a valid assumption in my experience), you can't move it into the `BankAccount` class. As a result, you've introduced a new subclass of `BankAccount` called `InterestBearingAccount`, and you've made `SavingsAccount` and `CheckingAccount` inherit from it.

The fact that both savings and checking accounts bear interest is indicated by the `InterestBearingAccount` pure virtual function `creditInterest`, which is presumably redefined in its subclasses `SavingsAccount` and `CheckingAccount`.

This new class hierarchy allows you to rewrite your loop as follows:

```
// better, but still not perfect
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  static_cast<InterestBearingAccount*>(*p)->creditInterest();


}
```

Although this loop still contains a nasty little cast, it's much more robust than it used to be, because it will continue to work even if new subclasses of `InterestBearingAccount` are added to your application.

To get rid of the cast entirely, you must make some additional changes to your design. One approach is to tighten up the specification of your list of accounts. If you could get a list of `InterestBearingAccount` objects instead of `BankAccount` objects, everything would be peachy:

```
// all interest-bearing accounts in the bank
list<InterestBearingAccount*> allIBAccounts;


// a loop that compiles and works, both now and forever
for (list<InterestBearingAccount*>::iterator p =
        allIBAccounts.begin();
     p != allIBAccounts.end();
     ++p) {


  (*p)->creditInterest();
```

```
}
```

If getting a more specialized list isn't an option, it might make sense to say that the `creditInterest` operation applies to all bank accounts, but that for non-interest-bearing accounts, it's just a no-op. That could be expressed this way:

```
class BankAccount {
public:
  virtual void creditInterest() {}


  ...


};


class SavingsAccount: public BankAccount { ... };


class CheckingAccount: public BankAccount { ... };


list<BankAccount*> allAccounts;


// look ma, no cast!
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  (*p)->creditInterest();


}
```

Notice that the virtual function `BankAccount::creditInterest` provides an empty default implementation. This is a convenient way to specify that its behavior is a no-op by default, but it can lead to unforeseen difficulties in its own right. For the inside story on why, as well as how to eliminate the danger, consult Item 36. Notice also that `creditInterest` is (implicitly) an inline function. There's nothing wrong with that, but because it's also virtual, the inline directive will probably be ignored. Item 33 explains why.

As you have seen, downcasts can be eliminated in a number of ways. The best way is to replace

such casts with calls to virtual functions, possibly also making each virtual function a no-op for any classes to which it doesn't truly apply. A second method is to tighten up the typing so that there is no ambiguity between the declared type of a pointer and the pointer type that you know is really there. Whatever the effort required to get rid of downcasts, it's effort well spent, because downcasts are ugly and error-prone, and they lead to code that's difficult to understand, enhance, and maintain (see Item M32).

What I've just written is the truth and nothing but the truth. It is not, however, the whole truth. There are occasions when you really do have to perform a downcast.

For example, suppose you faced the situation we considered at the outset of this Item, i.e., `allAccounts` holds `BankAccount` pointers, `creditInterest` is defined only for `SavingsAccount` objects, and you must write a loop to credit interest to every account. Further suppose that all those things are beyond your control; you can't change the definitions for `BankAccount`, `SavingsAccount`, or `allAccounts`. (This would happen if they were defined in a library to which you had read-only access.) If that were the case, you'd have to use downcasting, no matter how distasteful you found the idea.

Nevertheless, there is a better way to do it than through a raw cast such as we saw above. The better way is called "safe downcasting," and it's implemented via C++'s `dynamic_cast` operator (see Item M2). When you use `dynamic_cast` on a pointer, the cast is attempted, and if it succeeds (i.e., if the dynamic type of the pointer (see Item 38) is consistent with the type to which it's being cast), a valid pointer of the new type is returned. If the `dynamic_cast` fails, the null pointer is returned.

Here's the banking example with safe downcasting added:

```
class BankAccount { ... };          // as at the beginning of
                                    // this Item



class SavingsAccount:               // ditto
  public BankAccount { ... };



class CheckingAccount:              // ditto again
  public BankAccount { ... };



list<BankAccount*> allAccounts;     // this should look
                                    // familiar...
```

```
void error(const string& msg);      // error-handling function;
                                     // see below


// well, ma, at least the casts are safe...
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {


  // try safe-downcasting *p to a SavingsAccount*; see
  // below for information on the definition of psa
  if (SavingsAccount *psa =
        dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }


  // try safe-downcasting it to a CheckingAccount
  else if (CheckingAccount *pca =
             dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }


  // uh oh — unknown account type
  else {
    error("Unknown account type!");
  }
}
```

This scheme is far from ideal, but at least you can detect when your downcasts fail, something that's impossible without the use of `dynamic_cast`. Note, however, that prudence dictates you also check for the case where *all* the downcasts fail. That's the purpose of the final `else` clause in the code above. With virtual functions, there'd be no need for such a test, because every virtual call must resolve to *some* function. When you start downcasting, however, all bets are off. If somebody added a new type of account to the hierarchy, for example, but failed to update the code above, all the downcasts would fail. That's why it's important you handle that possibility. In all likelihood, it's not supposed to be the case that all the casts can fail, but when you allow downcasting, bad things start to happen to good programmers.

Did you check your glasses in a panic when you noticed what looks like variable definitions in the conditions of the `if` statements above? If so, worry not; your vision's fine. The ability to define such variables was added to the language at the same time as `dynamic_cast`. This feature lets you write neater code, because you don't really need `psa` or `pca` unless the `dynamic_casts` that initialize them

succeed, and with the new syntax, you don't have to define those variables outside the conditionals containing the casts. (Item 32 explains why you generally want to avoid superfluous variable definitions, anyway.) If your compilers don't yet support this new way of defining variables, you can do it the old way:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {



  SavingsAccount *psa;        // traditional definition
  CheckingAccount *pca;       // traditional definition


  if (psa = dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }


  else if (pca = dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }


  else {
    error("Unknown account type!");
  }
}
```

In the grand scheme of things, of course, where you place your definitions for variables like psa and pca is of little consequence. The important thing is this: the `if-then-else` style of programming that downcasting invariably leads to is vastly inferior to the use of virtual functions, and you should reserve it for situations in which you truly have no alternative. With any luck, you will never face such a bleak and desolate programming landscape.

Back to Item 38: Never redefine an inherited default parameter value.
Continue to Item 40: Model "has-a" or "is-implemented-in-terms-of" through layering.

Back to Item 39: Avoid casts down the inheritance hierarchy.
Continue to Item 41: Differentiate between inheritance and templates.

Item 40: Model "has-a" or "is-implemented-in-terms-of" through layering.

Layering is the process of building one class on top of another class by having the layering class contain an object of the layered class as a data member. For example:

```
class Address { ... };          // where someone lives


class PhoneNumber { ... };


class Person {
public:
  ...


private:
  string name;                  // layered object
  Address address;              // ditto
  PhoneNumber voiceNumber;      // ditto
  PhoneNumber faxNumber;        // ditto
};
```

In this example, the `Person` class is said to be layered on top of the `string`, `Address`, and `PhoneNumber` classes, because it contains data members of those types. The term *layering* has lots of synonyms. It's also known as *composition*, *containment*, and *embedding*.

Item 35 explains that public inheritance means "isa." In contrast, layering means either "has-a" or "is-implemented-in-terms-of."

The `Person` class above demonstrates the has-a relationship. A `Person` object has a name, an address, and telephone numbers for voice and FAX communication. You wouldn't say that a person *is* a name or that a person *is* an address. You would say that a person *has* a name and *has* an address, etc. Most people have little difficulty with this distinction, so confusion between the roles of isa and has-a is relatively rare.

Somewhat more troublesome is the difference between isa and is-implemented-in-terms-of. For example, suppose you need a template for classes representing sets of arbitrary objects, i.e.,

collections without duplicates. Because reuse is a wonderful thing, and because you wisely read Item 49's overview of the standard C++ library, your first instinct is to employ the library's `set` template. After all, why write a new template when you can use an established one written by somebody else?

As you delve into `set`'s documentation, however, you discover a limitation your application can't live with: a `set` requires that the elements contained within it be *totally ordered*, i.e., for every pair of objects `a` and `b` in the set, it must be possible to determine whether `a<b` or `b<a`. For many types, this requirement is easy to satisfy, and having a total ordering among objects allows `set` to offer certain attractive guarantees regarding its performance. (See Item 49 for more on performance guarantees in the standard library.) Your need, however, is for something more general: a `set`-like class where objects need not be totally ordered, they need only be what the C++ standard colorfully terms "EqualityComparable": it's possible to determine whether `a==b` for objects `a` and `b` of the same type. This more modest requirement is better suited to types representing things like colors. Is red less than green or is green less than red? For your application, it seems you'll need to write your own template after all.

Still, reuse is a wonderful thing. Being the data structure maven you are, you know that of the nearly limitless choices for implementing sets, one particularly simple way is to employ linked lists. But guess what? The `list` template (which generates linked list classes) is just *sitting* there in the standard library! You decide to (re)use it.

In particular, you decide to have your nascent `Set` template inherit from `list`. That is, `Set<T>` will inherit from `list<T>`. After all, in your implementation, a `Set` object will in fact *be* a `list` object. You thus declare your `Set` template like this:

```
// the wrong way to use list for Set
template<class T>
class Set: public list<T> { ... };
```

Everything may seem fine and dandy at this point, but in fact there is something quite wrong. As Item 35 explains, if D isa B, everything true of B is also true of D. However, a `list` object may contain duplicates, so if the value 3051 is inserted into a `list<int>` twice, that list will contain two copies of 3051. In contrast, a `Set` may not contain duplicates, so if the value 3051 is inserted into a `Set<int>` twice, the set contains only one copy of the value. It is thus a vicious lie that a `Set` isa `list`, because some of the things that are true for `list` objects are not true for `Set` objects.

Because the relationship between these two classes isn't isa, public inheritance is the wrong way to model that relationship. The right way is to realize that a `Set` object can be *implemented in terms of* a `list` object:

```
// the right way to use list for Set
```

```
template<class T>
class Set {
public:
  bool member(const T& item) const;


  void insert(const T& item);
  void remove(const T& item);


  int cardinality() const;


private:
  list<T> rep;                        // representation for a set
};
```

`Set`'s member functions can lean heavily on functionality already offered by `list` and other parts of the standard library, so the implementation is neither difficult to write nor thrilling to read:

```
template<class T>
bool Set<T>::member(const T& item) const
{ return find(rep.begin(), rep.end(), item) != rep.end(); }


template<class T>
void Set<T>::insert(const T& item)
{ if (!member(item)) rep.push_back(item); }


template<class T>
void Set<T>::remove(const T& item)
{
  list<T>::iterator it =
    find(rep.begin(), rep.end(), item);


  if (it != rep.end()) rep.erase(it);
}


template<class T>
int Set<T>::cardinality() const
{ return rep.size(); }
```

These functions are simple enough that they make reasonable candidates for inlining, though I

know you'd want to review the discussion in Item 33 before making any firm inlining decisions. (In the code above, functions like `find`, `begin`, `end`, `push_back`, etc., are part of the standard library's framework for working with container templates like `list`. You'll find an overview of this framework in Item 49 and M35.)

It's worth remarking that the `Set` class interface fails the test of being complete and minimal (see Item 18). In terms of completeness, the primary omission is that of a way to iterate over the contents of a set, something that might well be necessary for many applications (and that is provided by all members of the standard library, including `set`). An additional drawback is that `Set` fails to follow the container class conventions embraced by the standard library (see Items 49 and M35), and that makes it more difficult to take advantage of other parts of the library when working with `Sets`.

Nits about `Set`'s interface, however, shouldn't be allowed to overshadow what `Set` got indisputably right: the relationship between `Set` and `list`. That relationship is not isa (though it initially looked like it might be), it's "is-implemented-in-terms-of," and the use of layering to implement that relationship is something of which any class designer may be justly proud.

Incidentally, when you use layering to relate two classes, you create a compile-time dependency between those classes. For information on why this should concern you, as well as what you can do to allay your worries, turn to Item 34.

Back to Item 39: Avoid casts down the inheritance hierarchy.
Continue to Item 41: Differentiate between inheritance and templates.

Back to Item 40: Differentiate between inheritance and templates.
Continue to Item 42: Use private inheritance judiciously.

Item 41: Differentiate between inheritance and templates.

Consider the following two design problems:

- Being a devoted student of Computer Science, you want to create classes representing stacks of objects. You'll need several different classes, because each stack must be homogeneous, i.e., it must have only a single type of object in it. For example, you might have a class for stacks of `int`s, a second class for stacks of `string`s, a third for stacks of stacks of `string`s, etc. You're interested only in supporting a minimal interface to the class (see Item 18), so you'll limit your operations to stack creation, stack destruction, pushing objects onto the stack, popping objects off the stack, and determining whether the stack is empty. For this exercise, you'll ignore the classes in the standard library (including `stack` — see Item 49), because you crave the experience of writing the code yourself. Reuse is a wonderful thing, but when your goal is a deep understanding of how something works, there's nothing quite like diving in and getting your hands dirty.
- Being a devoted feline aficionado, you want to design classes representing cats. You'll need several different classes, because each breed of cat is a little different. Like all objects, cats can be created and destroyed, but, as any cat-lover knows, the only other things cats do are eat and sleep. However, each breed of cat eats and sleeps in its own endearing way.

These two problem specifications sound similar, yet they result in utterly different software designs. Why?

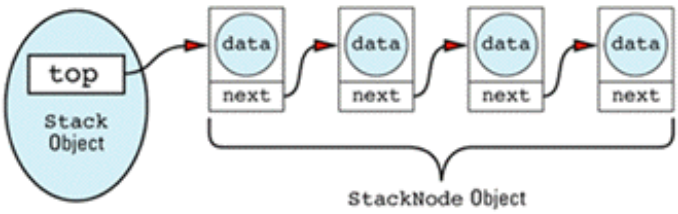
The answer has to do with the relationship between each class's behavior and the *type* of object being manipulated. With both stacks and cats, you're dealing with a variety of different types (stacks containing objects of type `T`, cats of breed `T`), but the question you must ask yourself is this: does the type `T` affect the *behavior* of the class? If `T` does *not* affect the behavior, you can use a template. If `T` *does* affect the behavior, you'll need virtual functions, and you'll therefore use inheritance.

Here's how you might define a linked-list implementation of a `Stack` class, assuming that the objects to be stacked are of type `T`:

```
class Stack {
public:
  Stack();
  ~Stack();


  void push(const T& object);
  T pop();
```

```
  bool empty() const;             // is stack empty?


private:
  struct StackNode {              // linked list node
    T data;                       // data at this node
    StackNode *next;              // next node in list


    // StackNode constructor initializes both fields
    StackNode(const T& newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };


  StackNode *top;                 // top of stack


  Stack(const Stack& rhs);                // prevent copying and
  Stack& operator=(const Stack& rhs);     // assignment (see Item 27)
};
```

`Stack` objects would thus build data structures that look like this:



The linked list itself is made up of `StackNode` objects, but that's an implementation detail of the `Stack` class, so `StackNode` has been declared a private type of `Stack`. Notice that `StackNode` has a constructor to make sure all its fields are initialized properly. Just because you can write linked lists in your sleep is no reason to omit technological advances such as constructors.

Here's a reasonable first cut at how you might implement the `Stack` member functions. As with many prototype implementations (and far too much production software), there's no checking for

errors, because in a prototypical world, nothing ever goes wrong.

```
Stack::Stack(): top(0) {}      // initialize top to null


void Stack::push(const T& object)
{
  top = new StackNode(object, top);    // put new node at
}                                      // front of list


T Stack::pop()
{
  StackNode *topOfStack = top;    // remember top node
  top = top->next;



  T data = topOfStack->data;      // remember node data
  delete topOfStack;



  return data;
}



Stack::~Stack()                   // delete all in stack
{
  while (top) {
    StackNode *toDie = top;       // get ptr to top node
    top = top->next;              // move to next node
    delete toDie;                 // delete former top node
  }
}


bool Stack::empty() const
{ return top == 0; }
```

There's nothing riveting about these implementations. In fact, the only interesting thing about them is this: you are able to write each member function knowing essentially *nothing* about `T`. (You assume you can call `T`'s copy constructor, but, as Item 45 explains, that's a pretty reasonable assumption.) The code you write for construction, destruction, pushing, popping, and determining whether the stack is empty is the same, no matter what `T` is. Except for the assumption that you can call `T`'s copy constructor, the behavior of a `stack` does not depend on `T` in any way. That's the

hallmark of a template class: the behavior doesn't depend on the type.

Turning your `Stack` class into a template, by the way, is so simple, even Dilbert's pointy-haired boss could do it:

```
template<class T> class Stack {

  ...                          // exactly the same as above

};
```

But on to cats. Why won't templates work with cats?

Reread the specification and note the requirement that "each breed of cat eats and sleeps in its own endearing way." That means you're going to have to implement *different behavior* for each type of cat. You can't just write a single function to handle all cats, all you can do is *specify an interface* for a function that each type of cat must implement. Aha! The way to propagate a function *interface only* is to declare a pure virtual function (see Item 36):

```
class Cat {
public:
  virtual ~Cat();                    // see Item 14

  virtual void eat() = 0;            // all cats eat
  virtual void sleep() = 0;          // all cats sleep
};
```

Subclasses of `Cat` — say, `Siamese` and `BritishShortHairedTabby` — must of course redefine the `eat` and `sleep` function interfaces they inherit:

```
class Siamese: public Cat {
public:
  void eat();
  void sleep();

  ...
```

```
};


class BritishShortHairedTabby: public Cat {
public:
  void eat();
  void sleep();


  ...


};
```

Okay, you now know why templates work for the `Stack` class and why they won't work for the `Cat` class. You also know why inheritance works for the `Cat` class. The only remaining question is why inheritance won't work for the `Stack` class. To see why, try to declare the root class of a `Stack` hierarchy, the single class from which all other stack classes would inherit:

```
class Stack {      // a stack of anything
public:
  virtual void push(const ??? object) = 0;
  virtual ??? pop() = 0;


  ...


};
```

Now the difficulty becomes clear. What types are you going to declare for the pure virtual functions `push` and `pop`? Remember that each subclass must redeclare the virtual functions it inherits with *exactly* the same parameter types and with return types consistent with the base class declarations. Unfortunately, a stack of `int`s will want to push and pop `int` objects, whereas a stack of, say, `Cat`s, will want to push and pop `Cat` objects. How can the `Stack` class declare its pure virtual functions in such a way that clients can create both stacks of `int`s and stacks of `Cat`s? The cold, hard truth is that it can't, and that's why inheritance is unsuitable for creating stacks.

But maybe you're the sneaky type. Maybe you think you can outsmart your compilers by using generic (`void*`) pointers. As it turns out, generic pointers don't help you here. You simply can't get around the requirement that a virtual function's declarations in derived classes must never contradict its declaration in the base class. However, generic pointers can help with a different

problem, one related to the efficiency of classes generated from templates. For details, see Item 42.

Now that we've dispensed with stacks and cats, we can summarize the lessons of this Item as follows:

- A template should be used to generate a collection of classes when the type of the objects *does not* affect the behavior of the class's functions.
- Inheritance should be used for a collection of classes when the type of the objects *does* affect the behavior of the class's functions.

Internalize these two little bullet points, and you'll be well on your way to mastering the choice between inheritance and templates.

Back to Item 40: Differentiate between inheritance and templates.
Continue to Item 42: Use private inheritance judiciously.

Back to Item 41: Differentiate between inheritance and templates.
Continue to Item 43: Use multiple inheritance judiciously.

Item 42: Use private inheritance judiciously.

Item 35 demonstrates that C++ treats public inheritance as an isa relationship. It does this by showing that compilers, when given a hierarchy in which a class `Student` publicly inherits from a class `Person`, implicitly convert `Students` to `Persons` when that is necessary for a function call to succeed. It's worth repeating a portion of that example using private inheritance instead of public inheritance:

```
class Person { ... };



class Student:                      // this time we use
  private Person { ... };           // private inheritance



void dance(const Person& p);        // anyone can dance


void study(const Student& s);       // only students study



Person p;                           // p is a Person
Student s;                          // s is a Student



dance(p);                           // fine, p is a Person



dance(s);                           // error! a Student isn't
                                    // a Person
```

Clearly, private inheritance doesn't mean isa. What does it mean then?

"Whoa!" you say. "Before we get to the meaning, let's cover the behavior. How does private inheritance behave?" Well, the first rule governing private inheritance you've just seen in action: in contrast to public inheritance, compilers will generally *not* convert a derived class object (such as

`Student`) into a base class object (such as `Person`) if the inheritance relationship between the classes is private. That's why the call to `dance` fails for the object `s`. The second rule is that members inherited from a private base class become private members of the derived class, even if they were protected or public in the base class. So much for behavior.

That brings us to meaning. Private inheritance means is-implemented-in-terms-of. If you make a class `D` privately inherit from a class `B`, you do so because you are interested in taking advantage of some of the code that has already been written for class `B`, not because there is any conceptual relationship between objects of type `B` and objects of type `D`. As such, private inheritance is purely an implementation technique. Using the terms introduced in Item 36, private inheritance means that implementation *only* should be inherited; interface should be ignored. If `D` privately inherits from `B`, it means that `D` objects are implemented in terms of `B` objects, nothing more. Private inheritance means nothing during software *design*, only during software *implementation*.

The fact that private inheritance means is-implemented-in-terms-of is a little disturbing, because Item 40 points out that layering can mean the same thing. How are you supposed to choose between them? The answer is simple: use layering whenever you can, use private inheritance whenever you must. When must you? When protected members and/or virtual functions enter the picture — but more on that in a moment.

Item 41 shows a way to write a `Stack` template that generates classes holding objects of different types. You may wish to familiarize yourself with that Item now. Templates are one of the most useful features in C++, but once you start using them regularly, you'll discover that if you instantiate a template a dozen times, you are likely to instantiate the *code* for the template a dozen times. In the case of the `Stack` template, the code making up `Stack<int>`'s member functions will be completely separate from the code making up `Stack<double>`'s member functions. Sometimes this is unavoidable, but such code replication is likely to exist even if the template functions could in fact share code. There is a name for the resultant increase in object code size: template-induced *code bloat*. It is not a good thing.

For certain kinds of classes, you can use generic pointers to avoid it. The classes to which this approach is applicable store *pointers* instead of objects, and they are implemented by:

1. Creating a single class that stores `void*` pointers to objects.
2. Creating a set of additional classes whose only purpose is to enforce strong typing. These classes all use the generic class of step 1 for the actual work.

Here's an example using the non-template `Stack` class of Item 41, except here it stores generic pointers instead of objects:

```
class GenericStack {
public:
```

```
  GenericStack();
  ~GenericStack();


  void push(void *object);
  void * pop();


  bool empty() const;


private:
  struct StackNode {
    void *data;                   // data at this node
    StackNode *next;              // next node in list


    StackNode(void *newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };


  StackNode *top;                           // top of stack


  GenericStack(const GenericStack& rhs);   // prevent copying and
  GenericStack&                            // assignment (see
    operator=(const GenericStack& rhs);    // Item 27)
};
```

Because this class stores pointers instead of objects, it is possible that an object is pointed to by more than one stack (i.e., has been pushed onto multiple stacks). It is thus of critical importance that `pop` and the class destructor *not* delete the `data` pointer of any `StackNode` object they destroy, although they must continue to delete the `StackNode` object itself. After all, the `StackNode` objects are allocated inside the `GenericStack` class, so they must also be deallocated inside that class. As a result, the implementation of the `Stack` class in Item 41 suffices almost perfectly for the `GenericStack` class. The only changes you need to make involve substitutions of `void*` for `T`.

The `GenericStack` class by itself is of little utility — it's too easy to misuse. For example, a client could mistakenly push a pointer to a `Cat` object onto a stack meant to hold only pointers to `int`s, and compilers would merrily accept it. After all, a pointer's a pointer when it comes to `void*` parameters.

To regain the type safety to which you have become accustomed, you create *interface classes* to `GenericStack`, like this:

```
class IntStack {                  // interface class for ints
public:
  void push(int *intPtr) { s.push(intPtr); }
  int * pop() { return static_cast<int*>(s.pop()); }
  bool empty() const { return s.empty(); }


private:
  GenericStack s;                 // implementation
};



class CatStack {                  // interface class for cats
public:
  void push(Cat *catPtr) { s.push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(s.pop()); }
  bool empty() const { return s.empty(); }


private:
  GenericStack s;                 // implementation
};
```

As you can see, the `IntStack` and `CatStack` classes serve only to enforce strong typing. Only `int` pointers can be pushed onto an `IntStack` or popped from it, and only `Cat` pointers can be pushed onto a `CatStack` or popped from it. Both `IntStack` and `CatStack` are implemented in terms of the class `GenericStack`, a relationship that is expressed through layering (see Item 40), and `IntStack` and `CatStack` will share the code for the functions in `GenericStack` that actually implement their behavior. Furthermore, the fact that all `IntStack` and `CatStack` member functions are (implicitly) `inline` means that the runtime cost of using these interface classes is zip, zero, nada, nil.

But what if potential clients don't realize that? What if they mistakenly believe that use of `GenericStack` is more efficient, or what if they're just wild and reckless and think only wimps need type-safety nets? What's to keep them from bypassing `IntStack` and `CatStack` and going straight to `GenericStack`, where they'll be free to make the kinds of type errors C++ was specifically designed to prevent?

Nothing. Nothing prevents that. But maybe something should.

I mentioned at the outset of this Item that an alternative way to assert an is-implemented-in-terms-

of relationship between classes is through private inheritance. In this case, that technique offers an advantage over layering, because it allows you to express the idea that `GenericStack` is too unsafe for general use, that it should be used only to implement other classes. You say that by protecting `GenericStack`'s member functions:

```
class GenericStack {
protected:
  GenericStack();
  ~GenericStack();



  void push(void *object);
  void * pop();



  bool empty() const;



private:
  ...                           // same as above
};



GenericStack s;                 // error! constructor is
                                // protected
```

```
class IntStack: private GenericStack {
public:
  void push(int *intPtr) { GenericStack::push(intPtr); }
  int * pop() { return static_cast<int*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

```
class CatStack: private GenericStack {
public:
  void push(Cat *catPtr) { GenericStack::push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

```
IntStack is;                    // fine


CatStack cs;                    // also fine
```

Like the layering approach, the implementation based on private inheritance avoids code duplication, because the type-safe interface classes consist of nothing but inline calls to the underlying `GenericStack` functions.

Building type-safe interfaces on top of the `GenericStack` class is a pretty slick maneuver, but it's awfully unpleasant to have to type in all those interface classes by hand. Fortunately, you don't have to. You can use templates to generate them automatically. Here's a template to generate type-safe stack interfaces using private inheritance:

```
template<class T>
class Stack: private GenericStack {
public:
  void push(T *objectPtr) { GenericStack::push(objectPtr); }
  T * pop() { return static_cast<T*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

This is amazing code, though you may not realize it right away. Because of the template, compilers will automatically generate as many interface classes as you need. Because those classes are type-safe, client type errors are detected during compilation. Because `GenericStack`'s member functions are protected and interface classes use it as a private base class, clients are unable to bypass the interface classes. Because each interface class member function is (implicitly) declared `inline`, no runtime cost is incurred by use of the type-safe classes; the generated code is exactly the same as if clients programmed with `GenericStack` directly (assuming compilers respect the `inline` request — see Item 33). And because `GenericStack` uses `void*` pointers, you pay for only one copy of the code for manipulating stacks, no matter how many different types of stack you use in your program. In short, this design gives you code that's both maximally efficient and maximally type safe. It's difficult to do better than that.

One of the precepts of this book is that C++'s features interact in remarkable ways. This example, I hope you'll agree, is pretty remarkable.

The insight to carry away from this example is that it could not have been achieved using layering. Only inheritance gives access to protected members, and only inheritance allows for virtual functions to be redefined. (For an example of how the existence of virtual functions can motivate the use of private inheritance, see Item 43.) Because virtual functions and protected members exist,

private inheritance is sometimes the only practical way to express an is-implemented-in-terms-of relationship between classes. As a result, you shouldn't be afraid to use private inheritance when it's the most appropriate implementation technique at your disposal. At the same time, however, layering is the preferable technique in general, so you should employ it whenever you can.

Back to Item 41: Differentiate between inheritance and templates.
Continue to Item 43: Use multiple inheritance judiciously.

Back to Item 42: Use private inheritance judiciously.
Continue to Item 44: Say what you mean; understand what you're saying.

Item 43: Use multiple inheritance judiciously.

Depending on who's doing the talking, multiple inheritance (MI) is either the product of divine inspiration or the manifest work of the devil. Proponents hail it as essential to the natural modeling of real-world problems, while critics argue that it is slow, difficult to implement, and no more powerful than single inheritance. Disconcertingly, the world of object-oriented programming languages remains split on the issue: C++, Eiffel, and the Common LISP Object System (CLOS) offer MI; Smalltalk, Objective C, and Object Pascal do not; and Java supports only a restricted form of it. What's a poor, struggling programmer to believe?

Before you believe anything, you need to get your facts straight. The one indisputable fact about MI in C++ is that it opens up a Pandora's box of complexities that simply do not exist under single inheritance. Of these, the most basic is ambiguity (see Item 26). If a derived class inherits a member name from more than one base class, any reference to that name is ambiguous; you must explicitly say which member you mean. Here's an example that's based on a discussion in the ARM (see Item 50):

```
class Lottery {
public:
  virtual int draw();


  ...


};


class GraphicalObject {
public:
  virtual int draw();


  ...


};



class LotterySimulation: public Lottery,
                         public GraphicalObject {
```

```
  ...                          // doesn't declare draw

};

LotterySimulation *pls = new LotterySimulation;

pls->draw();                   // error! — ambiguous
pls->Lottery::draw();          // fine
pls->GraphicalObject::draw();  // fine
```

This looks clumsy, but at least it works. Unfortunately, the clumsiness is difficult to eliminate. Even if one of the inherited `draw` functions were private and hence inaccessible, the ambiguity would remain. (There's a good reason for that, but a complete explanation of the situation is provided in Item 26, so I won't repeat it here.)

Explicitly qualifying members is more than clumsy, however, it's also limiting. When you explicitly qualify a virtual function with a class name, the function doesn't act virtual any longer. Instead, the function called is precisely the one you specify, even if the object on which it's invoked is of a derived class:

```
class SpecialLotterySimulation: public LotterySimulation {
public:
  virtual int draw();

  ...

};

pls = new SpecialLotterySimulation;

pls->draw();                       // error! — still ambiguous
pls->Lottery::draw();              // calls Lottery::draw
pls->GraphicalObject::draw();      // calls GraphicalObject::draw
```

In this case, notice that even though `pls` points to a `SpecialLotterySimulation` object, there is no way (short of a downcast — see Item 39) to invoke the `draw` function defined in that class.

But wait, there's more. The `draw` functions in both `Lottery` and `GraphicalObject` are declared virtual so that subclasses can redefine them (see Item 36), but what if `LotterySimulation` would like to redefine *both* of them? The unpleasant truth is that it can't, because a class is allowed to have only a single function called `draw` that takes no arguments. (There is a special exception to this rule if one of the functions is `const` and one is not — see Item 21.)

At one point, this difficulty was considered a serious enough problem to justify a change in the language. The ARM discusses the possibility of allowing inherited virtual functions to be "renamed," but then it was discovered that the problem can be circumvented by the addition of a pair of new classes:

```
class AuxLottery: public Lottery {
public:
  virtual int lotteryDraw() = 0;


  virtual int draw() { return lotteryDraw(); }
};


class AuxGraphicalObject: public GraphicalObject {
public:
  virtual int graphicalObjectDraw() = 0;


  virtual int draw() { return graphicalObjectDraw(); }
};



class LotterySimulation: public AuxLottery,
                         public AuxGraphicalObject {
public:
  virtual int lotteryDraw();
  virtual int graphicalObjectDraw();


  ...


};
```

Each of the two new classes, `AuxLottery` and `AuxGraphicalObject`, essentially declares a new name for the `draw` function that each inherits. This new name takes the form of a pure virtual function, in this case `lotteryDraw` and `graphicalObjectDraw`; the functions are pure virtual so that concrete subclasses must redefine them. Furthermore, each class redefines the `draw` that it inherits to itself invoke the new pure virtual function. The net effect is that within this class hierarchy, the single, ambiguous name `draw` has effectively been split into two unambiguous, but operationally equivalent, names: `lotteryDraw` and `graphicalObjectDraw`:

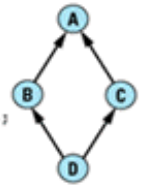
```
LotterySimulation *pls = new LotterySimulation;


Lottery *pl = pls;
GraphicalObject *pgo = pls;


// this calls LotterySimulation::lotteryDraw
pl->draw();


// this calls LotterySimulation::graphicalObjectDraw
pgo->draw();
```

This strategy, replete as it is with the clever application of pure virtual, simple virtual, and inline functions (see Item 33), should be committed to memory. In the first place, it solves a problem that you may encounter some day. In the second, it can serve to remind you of the complications that can arise in the presence of multiple inheritance. Yes, this tactic works, but do you really want to be forced to introduce new classes just so you can redefine a virtual function? The classes `AuxLottery` and `AuxGraphicalObject` are essential to the correct operation of this hierarchy, but they correspond neither to an abstraction in the problem domain nor to an abstraction in the implementation domain. They exist purely as an implementation device — nothing more. You already know that good software is "device independent." That rule of thumb applies here, too.

The ambiguity problem, interesting though it is, hardly begins to scratch the surface of the issues you'll confront when you flirt with MI. Another one grows out of the empirical observation that an inheritance hierarchy that starts out looking like this,

has a distressing tendency to evolve into one that looks like this:

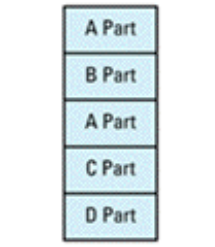
```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```



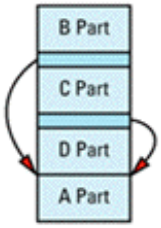
Now, it may or may not be true that diamonds are a girl's best friend, but it is certainly true that a diamond-shaped inheritance hierarchy such as this is *not* very friendly. If you create a hierarchy such as this, you are immediately confronted with the question of whether to make `A` a virtual base class, i.e., whether inheritance from A should be virtual. In practice, the answer is almost invariably that it should; only rarely will you want an object of type `D` to contain multiple copies of the data members of `A`. In recognition of this truth, `B` and `C` above declare `A` as a virtual base class.

Unfortunately, at the time you define `B` and `C`, you may not know whether any class will decide to inherit from both of them, and in fact you shouldn't need to know this in order to define them correctly. As a class designer, this puts you in a dreadful quandary. If you do *not* declare `A` as a virtual base of `B` and `C`, a later designer of `D` may need to modify the definitions of `B` and `C` in order to use them effectively. Frequently, this is unacceptable, often because the definitions of `A`, `B`, and `C` are read-only. This would be the case if `A`, `B`, and `C` were in a library, for example, and `D` was written by a library client.

On the other hand, if you *do* declare `A` as a virtual base of `B` and `C`, you typically impose an additional cost in both space and time on clients of those classes. That is because virtual base classes are often implemented as *pointers* to objects, rather than as objects themselves. It goes without saying that the layout of objects in memory is compiler-dependent, but the fact remains that the memory layout for an object of type `D` with `A` as a nonvirtual base is typically a contiguous series of memory locations, whereas the memory layout for an object of type `D` with `A` as a virtual base is sometimes a contiguous series of memory locations, two of which contain pointers to the memory locations containing the data members of the virtual base class:



Common memory layout of a **D** object where **A** is a nonvirtual base class



Some compilers' memory layout of a **D** object where **A** is a virtual base class

Even compilers that don't use this particular implementation strategy generally impose some kind of space penalty for using virtual inheritance.

In view of these considerations, it would seem that effective class design in the presence of MI calls for clairvoyance on the part of library designers. Seeing as how run-of-the-mill common sense is an increasingly rare commodity these days, you would be ill-advised to rely too heavily on a language feature that calls for designers to be not only anticipatory of future needs, but downright prophetic (see also M32).

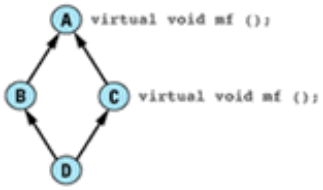
Of course, this could also be said of the choice between virtual and nonvirtual functions in a base class, but there is a crucial difference. Item 36 explains that a virtual function has a well-defined high-level meaning that is distinct from the equally well-defined high-level meaning of a nonvirtual function, so it is possible to choose between the two based on what you want to communicate to writers of subclasses. However, the decision whether a base class should be virtual or nonvirtual lacks a well-defined high-level meaning. Rather, that decision is usually based on the structure of the entire inheritance hierarchy, and as such it cannot be made until the entire hierarchy is known. If you need to know exactly how your class is going to be used before you can define it correctly, it becomes very difficult to design effective classes.

Once you're past the problem of ambiguity and you've settled the question of whether inheritance from your base class(es) should be virtual, still more complications confront you. Rather than belaboring things, I'll simply mention two other issues you need to keep in mind:

- **Passing constructor arguments to virtual base classes.** Under nonvirtual inheritance, arguments for a base class constructor are specified in the member initialization lists of the classes that are immediately derived from the base class. Because single inheritance hierarchies need only nonvirtual bases, arguments are passed up the inheritance hierarchy in a very natural fashion: the classes at level n of the hierarchy pass arguments to the classes at level n-1. For constructors of a virtual base class, however, arguments are specified in the

member initialization lists of the classes that are *most derived* from the base. As a result, the class initializing a virtual base may be arbitrarily far from it in the inheritance graph, and the class performing the initialization can change as new classes are added to the hierarchy. (A good way to avoid this problem is to eliminate the need to pass constructor arguments to virtual bases. The easiest way to do that is to avoid putting data members in such classes. This is the essence of the Java solution to the problem: virtual base classes in Java (i.e., "Interfaces") are prohibited from containing data.)

- **Dominance of virtual functions.** Just when you thought you had ambiguity all figured out, they change the rules on you. Consider again the diamond-shaped inheritance graph involving classes `A`, `B`, `C`, and `D`. Suppose that `A` defines a virtual member function `mf`, and `C` redefines it; `B` and `D`, however, do not redefine `mf`:



From our earlier discussion, you'd expect this to be ambiguous:

```
D *pd = new D;
pd->mf();                       // A::mf or C::mf?
```

Which `mf` should be called for a `D` object, the one directly inherited from `C` or the one indirectly inherited (via `B`) from `A`? The answer is that *it depends on how `B` and `C` inherit from `A`*. In particular, if `A` is a nonvirtual base of `B` or `C`, the call is ambiguous, but if `A` is a virtual base of both `B` and `C`, the redefinition of `mf` in `C` is said to *dominate* the original definition in `A`, and the call to `mf` through `pd` will resolve (unambiguously) to `C::mf`. If you sit down and work it all out, it emerges that this is the behavior you want, but it's kind of a pain to have to sit down and work it all out before it makes sense.

Perhaps by now you agree that MI can lead to complications. Perhaps you are convinced that no one in their right mind would ever use it. Perhaps you are prepared to propose to the international ◦C++ standardization committee that multiple inheritance be removed from the language, or at least to propose to your manager that programmers at your company be physically barred from using it.

Perhaps you are being too hasty.

Bear in mind that the designer of C++ didn't set out to make multiple inheritance hard to use, it just turned out that making all the pieces work together in a more or less reasonable fashion inherently

entailed the introduction of certain complexities. In the above discussion, you may have noticed that the bulk of these complexities arise in conjunction with the use of virtual base classes. If you can avoid the use of virtual bases — that is, if you can avoid the creation of the deadly diamond inheritance graph — things become much more manageable.

For example, Item 34 describes how a *Protocol class* exists only to specify an interface for derived classes; it has no data members, no constructors, a virtual destructor (see Item 14), and a set of pure virtual functions that specify the interface. A Protocol `Person` class might look like this:

```
class Person {
public:
  virtual ~Person();


  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

Clients of this class must program in terms of `Person` pointers and references, because abstract classes cannot be instantiated.

To create objects that can be manipulated as `Person` objects, clients of `Person` use *factory functions* (see Item 34) to instantiate concrete subclasses of that class:

```
// factory function to create a Person object from a
// unique database ID
Person * makePerson(DatabaseID personIdentifier);


DatabaseID askUserForDatabaseID();



DatabaseID pid = askUserForDatabaseID();



Person *pp = makePerson(pid);    // create object supporting
                                 // the Person interface



...                              // manipulate *pp via
                                 // Person's member functions
```

```
delete pp;                          // delete the object when
                                    // it's no longer needed
```

This just begs the question: how does `makePerson` create the objects to which it returns pointers? Clearly, there must be some concrete class derived from `Person` that `makePerson` can instantiate.

Suppose this class is called `MyPerson`. As a concrete class, `MyPerson` must provide implementations for the pure virtual functions it inherits from `Person`. It could write these from scratch, but it would be better software engineering to take advantage of existing components that already do most or all of what's necessary. For example, let's suppose a creaky old database-specific class `PersonInfo` already exists that provides the essence of what `MyPerson` needs:

```
class PersonInfo {
public:
  PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();


  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;


  virtual const char * valueDelimOpen() const;       // see
  virtual const char * valueDelimClose() const;      // below


  ...


};
```

You can tell this is an old class, because the member functions return `const char*`s instead of `string` objects. Still, if the shoe fits, why not wear it? The names of this class's member functions suggest that the result is likely to be pretty comfortable.

You come to discover that `PersonInfo`, however, was designed to facilitate the process of printing database fields in various formats, with the beginning and end of each field value delimited by special strings. By default, the opening and closing delimiters for field values are braces, so the field value "Ring-tailed Lemur" would be formatted this way:

```
[Ring-tailed Lemur]
```

In recognition of the fact that braces are not universally desired by clients of `PersonInfo`, the virtual functions `valueDelimOpen` and `valueDelimClose` allow derived classes to specify their own opening and closing delimiter strings. The implementations of `PersonInfo`'s `theName`, `theBirthDate`, `theAddress`, and `theNationality` call these virtual functions to add the appropriate delimiters to the values they return. Using `PersonInfo::name` as an example, the code looks like this:

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                   // default opening delimiter
}


const char * PersonInfo::valueDelimClose() const
{
  return "]";                   // default closing delimiter
}


const char * PersonInfo::theName() const
{
  // reserve buffer for return value. Because this is
  // static, it's automatically initialized to all zeros
  static char value[MAX_FORMATTED_FIELD_VALUE_LENGTH];


  // write opening delimiter
  strcpy(value, valueDelimOpen());


  append to the string in value this object's name field


  // write closing delimiter
  strcat(value, valueDelimClose());


  return value;
}
```

One might quibble with the design of `PersonInfo::theName` (especially the use of a fixed-size static buffer — see Item 23), but set your quibbles aside and focus instead on this: `theName` calls

valueDelimOpen to generate the opening delimiter of the string it will return, then it generates the name value itself, then it calls valueDelimClose. Because valueDelimOpen and valueDelimClose are virtual functions, the result returned by theName is dependent not only on PersonInfo, but also on the classes derived from PersonInfo.

As the implementer of MyPerson, that's good news, because while perusing the fine print in the Person documentation, you discover that name and its sister member functions are required to return unadorned values, i.e., no delimiters are allowed. That is, if a person is from Madagascar, a call to that person's nationality function should return "Madagascar", not "[Madagascar]".

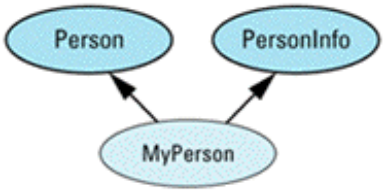
The relationship between MyPerson and PersonInfo is that PersonInfo happens to have some functions that make MyPerson easier to implement. That's all. There's no isa or has-a relationship anywhere in sight. Their relationship is thus is-implemented-in-terms-of, and we know that can be represented in two ways: via layering (see Item 40) and via private inheritance (see Item 42). Item 42 points out that layering is the generally preferred approach, but private inheritance is necessary if virtual functions are to be redefined. In this case, MyPerson needs to redefine valueDelimOpen and valueDelimClose, so layering won't do and private inheritance it must be: MyPerson must privately inherit from PersonInfo.

But MyPerson must also implement the Person interface, and that calls for public inheritance. This leads to one reasonable application of multiple inheritance: combine public inheritance of an interface with private inheritance of an implementation:

```
class Person {                          // this class specifies
public:                                 // the interface to be
  virtual ~Person();                    // implemented


  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};



class DatabaseID { ... };               // used below; details
                                        // are unimportant



class PersonInfo {                      // this class has functions
public:                                 // useful in implementing
  PersonInfo(DatabaseID pid);           // the Person interface
  virtual ~PersonInfo();
```

```
  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;



  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;



  ...



};



class MyPerson: public Person,        // note use of
                private PersonInfo {  // multiple inheritance
public:
  MyPerson(DatabaseID pid): PersonInfo(pid) {}



  // redefinitions of inherited virtual delimiter functions
  const char * valueDelimOpen() const { return ""; }
  const char * valueDelimClose() const { return ""; }



  // implementations of the required Person member functions
  string name() const
  { return PersonInfo::theName(); }



  string birthDate() const
  { return PersonInfo::theBirthDate(); }



  string address() const
  { return PersonInfo::theAddress(); }



  string nationality() const
  { return PersonInfo::theNationality(); }
};
```

Graphically, it looks like this:



This kind of example demonstrates that MI can be both useful and comprehensible, although it's no accident that the dreaded diamond-shaped inheritance graph is conspicuously absent.

Still, you must guard against temptation. Sometimes you can fall into the trap of using MI to make a quick fix to an inheritance hierarchy that would be better served by a more fundamental redesign. For example, suppose you're working with a hierarchy for animated cartoon characters. At least conceptually, it makes sense for any kind of character to dance and sing, but the way in which each type of character performs these activities differs. Furthermore, the default behavior for singing and dancing is to do nothing.

The way to say all that in C++ is like this:

```
class CartoonCharacter {
public:
  virtual void dance() {}
  virtual void sing() {}
};
```

Virtual functions naturally model the constraint that dancing and singing make sense for all `CartoonCharacter` objects. Do-nothing default behavior is expressed by the empty definitions of those functions in the class (see Item 36). Suppose a particular type of cartoon character is a grasshopper, which dances and sings in its own particular way:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();    // definition is elsewhere
  virtual void sing();     // definition is elsewhere
};
```

Now suppose that after implementing the `Grasshopper` class, you decide you also need a class for crickets:

```
class Cricket: public CartoonCharacter {
public:
```

```
  virtual void dance();
  virtual void sing();
};
```

As you sit down to implement the `Cricket` class, you realize that a lot of the code you wrote for the `Grasshopper` class can be reused. However, it needs to be tweaked a bit here and there to account for the differences in singing and dancing between grasshoppers and crickets. You are suddenly struck by a clever way to reuse your existing code: you'll implement the `Cricket` class *in terms of* the `Grasshopper` class, and you'll use virtual functions to allow the `Cricket` class to customize `Grasshopper` behavior!

You immediately recognize that these twin requirements — an is-implemented-in-terms-of relationship and the ability to redefine virtual functions — mean that `Cricket` will have to privately inherit from `Grasshopper`, but of course a cricket is still a cartoon character, so you redefine `Cricket` to inherit from both `Grasshopper` and `CartoonCharacter`:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance();
  virtual void sing();
};
```

You then set out to make the necessary modifications to the `Grasshopper` class. In particular, you need to declare some new virtual functions for `Cricket` to redefine:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();


protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();


  virtual void singCustomization();
};
```

Dancing for grasshoppers is now defined like this:

```
void Grasshopper::dance()
{
  perform common dancing actions;
```

```
  danceCustomization1();


  perform more common dancing actions;


  danceCustomization2();


  perform final common dancing actions;
}
```

Grasshopper singing is similarly orchestrated.

Clearly, the `Cricket` class must be updated to take into account the new virtual functions it must redefine:

```
class Cricket:public CartoonCharacter,
      private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }


protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();


  virtual void singCustomization();
};
```

This seems to work fine. When a `Cricket` object is told to dance, it will execute the common `dance` code in the `Grasshopper` class, then execute the `dance` customization code in the `Cricket` class, then continue with the code in `Grasshopper::dance`, etc.

There is a serious flaw in your design, however, and that is that you have run headlong into Occam's razor, a bad idea with a razor of any kind, and especially so when it belongs to William of Occam. Occamism preaches that entities should not be multiplied beyond necessity, and in this case, the entities in question are inheritance relationships. If you believe that multiple inheritance is more complicated than single inheritance (and I hope that you do), then the design of the `Cricket`

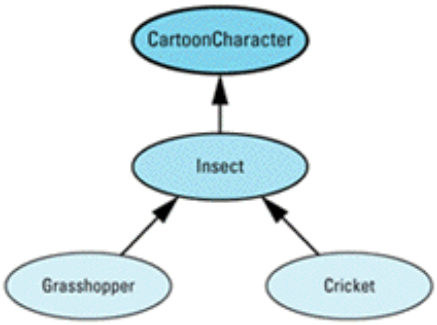
class is needlessly complex.

Fundamentally, the problem is that it is *not* true that the `Cricket` class is-implemented-in-terms-of the `Grasshopper` class. Rather, the `Cricket` class and the `Grasshopper` class *share common code*. In particular, they share the code that determines the dancing and singing behavior that grasshoppers and crickets have in common.

The way to say that two classes have something in common is not to have one class inherit from the other, but to have *both* of them inherit from a common base class. The common code for grasshoppers and crickets doesn't belong in the `Grasshopper` class, nor does it belong in the `Cricket` class. It belongs in a new class from which they both inherit, say, `Insect`:

```
class CartoonCharacter { ... };


class Insect: public CartoonCharacter {
public:
  virtual void dance();    // common code for both
  virtual void sing();     // grasshoppers and crickets


protected:
  virtual void danceCustomization1() = 0;
  virtual void danceCustomization2() = 0;


  virtual void singCustomization() = 0;
};


class Grasshopper: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();


  virtual void singCustomization();
};


class Cricket: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
```

```
  virtual void singCustomization();
};
```



Notice how much cleaner this design is. Only single inheritance is involved, and furthermore, only *public* inheritance is used. `Grasshopper` and `Cricket` define only customization functions; they inherit the `dance` and `sing` functions unchanged from `Insect`. William of Occam would be proud.

Although this design is cleaner than the one involving MI, it may initially have appeared to be inferior. After all, compared to the MI approach, this single-inheritance architecture calls for the introduction of a brand new class, a class unnecessary if MI is used. Why introduce an extra class if you don't have to?

This brings you face to face with the seductive nature of multiple inheritance. On the surface, MI seems to be the easier course of action. It adds no new classes, and though it calls for the addition of some new virtual functions to the `Grasshopper` class, those functions have to be added somewhere in any case.

Imagine now a programmer maintaining a large C++ class library, one in which a new class has to be added, much as the `Cricket` class had to be added to the existing `CartoonCharacter`/`Grasshopper` hierarchy. The programmer knows that a large number of clients use the existing hierarchy, so the bigger the change to the library, the greater the disruption to clients. The programmer is determined to minimize that kind of disruption. Mulling over the options, the programmer realizes that if a single private inheritance link from `Grasshopper` to `Cricket` is added, no other change to the hierarchy will be needed. The programmer smiles at the thought, pleased with the prospect of a large increase in functionality at the cost of only a slight increase in complexity.

Imagine now that that maintenance programmer is you. Resist the seduction.

Back to Item 42: Use private inheritance judiciously.
Continue to Item 44: Say what you mean; understand what you're saying.

Back to Item 43: Use multiple inheritance judiciously.
Continue to Miscellany

Item 44: Say what you mean; understand what you're saying.

In the introduction to this section on inheritance and object-oriented design, I emphasized the importance of understanding what different object-oriented constructs in C++ *mean*. This is quite different from just knowing the rules of the language. For example, the rules of C++ say that if class `D` publicly inherits from class `B`, there is a standard conversion from a `D` pointer to a `B` pointer; that the public member functions of `B` are inherited as public member functions of `D`, etc. That's all true, but it's close to useless if you're trying to translate your design into C++. Instead, you need to understand that public inheritance means isa, that if `D` publicly inherits from `B`, every object of type `D` isa object of type `B`, too. Thus, if you mean isa in your design, you know you have to use public inheritance.

Saying what you mean is only half the battle. The flip side of the coin is understanding what you're saying, and it's just as important. For example, it's irresponsible, if not downright immoral, to run around declaring member functions nonvirtual without recognizing that in so doing you are imposing constraints on subclasses. In declaring a nonvirtual member function, what you're really saying is that the function represents an invariant over specialization, and it would be disastrous if you didn't know that.

The equivalence of public inheritance and isa, and of nonvirtual member functions and invariance over specialization, are examples of how certain C++ constructs correspond to design-level ideas. The list below summarizes the most important of these mappings.

- **A common base class means common traits.** If class `D1` and class `D2` both declare class `B` as a base, `D1` and `D2` inherit common data members and/or common member functions from `B`. See Item 43.
- **Public inheritance means isa.** If class `D` publicly inherits from class `B`, every object of type `D` is also an object of type `B`, but not vice versa. See Item 35.
- **Private inheritance means is-implemented-in-terms-of.** If class `D` privately inherits from class `B`, objects of type `D` are simply implemented in terms of objects of type `B`; no conceptual relationship exists between objects of types `B` and `D`. See Item 42.
- **Layering means has-a or is-implemented-in-terms-of.** If class `A` contains a data member of type `B`, objects of type `A` either have a component of type `B` or are implemented in terms of objects of type `B`. See Item 40.

The following mappings apply only when public inheritance is involved:

- **A pure virtual function means that only the function's interface is inherited.** If a class `C` declares a pure virtual member function `mf`, subclasses of `C` must inherit the interface for `mf`, and concrete subclasses of `C` must supply their own implementations for it. See Item 36.

- **A simple virtual function means that the function's interface plus a default implementation is inherited.** If a class `C` declares a simple (not pure) virtual function `mf`, subclasses of `C` must inherit the interface for `mf`, and they may also inherit a default implementation, if they choose. See Item 36.
- **A nonvirtual function means that the function's interface plus a mandatory implementation is inherited.** If a class `C` declares a nonvirtual member function `mf`, subclasses of `C` must inherit both the interface for `mf` and its implementation. In effect, `mf` defines an invariant over specialization of `C`. See Item 36.

Back to Item 43: Use multiple inheritance judiciously.
Continue to Miscellany

Back to Item 44: Say what you mean; understand what you're saying.
Continue to Item 45: Know what functions C++ silently writes and calls.

Miscellany

Some guidelines for effective C++ programming defy convenient categorization. This section is where such guidelines come to roost. Not that that diminishes their importance. If you are to write effective software, you must understand what compilers are doing for you (to you?) behind your back, how to ensure that non-local static objects are initialized before they are used, what you can expect from the standard library, and where to go for insights into the language's underlying design philosophy. In this final section of the book, I expound on these issues, and more.

Back to Item 44: Say what you mean; understand what you're saying.
Continue to Item 45: Know what functions C++ silently writes and calls.

Back to Miscellany
Continue to Item 46: Prefer compile-time and link-time errors to runtime errors.

Item 45: Know what functions C++ silently writes and calls.

When is an empty class not an empty class? When C++ gets through with it. If you don't declare them yourself, your thoughtful compilers will declare their own versions of a copy constructor, an assignment operator, a destructor, and a pair of address-of operators. Furthermore, if you don't declare any constructors, they will declare a default constructor for you, too. All these functions will be public. In other words, if you write this,

```
class Empty{};
```

it's the same as if you'd written this:

```
class Empty {
public:
  Empty();                          // default constructor
  Empty(const Empty& rhs);          // copy constructor

  ~Empty();                         // destructor — see
                                    // below for whether
                                    // it's virtual
  Empty&
  operator=(const Empty& rhs);      // assignment operator

  Empty* operator&();               // address-of operators
  const Empty* operator&() const;
};
```

Now these functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:

```
const Empty e1;                     // default constructor;
                                    // destructor

Empty e2(e1);                       // copy constructor
```

```
e2 = e1;                              // assignment operator


Empty *pe2 = &e2;                     // address-of
                                      // operator (non-const)


const Empty *pe1 = &e1;               // address-of
                                      // operator (const)
```

Given that compilers are writing functions for you, what do the functions do? Well, the default constructor and the destructor don't really do anything. They just enable you to create and destroy objects of the class. (They also provide a convenient place for implementers to place code whose execution takes care of "behind the scenes" behavior — see Items 33 and M24.) Note that the generated destructor is nonvirtual (see Item 14) unless it's for a class inheriting from a base class that itself declares a virtual destructor. The default address-of operators just return the address of the object. These functions are effectively defined like this:

```
inline Empty::Empty() {}


inline Empty::~Empty() {}


inline Empty * Empty::operator&() { return this; }


inline const Empty * Empty::operator&() const
{ return this; }
```

As for the copy constructor and the assignment operator, the official rule is this: the default copy constructor (assignment operator) performs memberwise copy construction (assignment) of the nonstatic data members of the class. That is, if `m` is a nonstatic data member of type `T` in a class `C` and `C` declares no copy constructor (assignment operator), `m` will be copy constructed (assigned) using the copy constructor (assignment operator) defined for `T`, if there is one. If there isn't, this rule will be recursively applied to `m`'s data members until a copy constructor (assignment operator) or built-in type (e.g., `int`, `double`, pointer, etc.) is found. By default, objects of built-in types are copy constructed (assigned) using bitwise copy from the source object to the destination object. For classes that inherit from other classes, this rule is applied to each level of the inheritance hierarchy,

so user-defined copy constructors and assignment operators are called at whatever level they are declared.

I hope that's crystal clear.

But just in case it's not, here's an example. Consider the definition of a `NamedObject` template, whose instances are classes allowing you to associate names with objects:

```
template<class T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const string& name, const T& value);


  ...


private:
  string nameValue;
  T objectValue;
};
```

Because the `NamedObject` classes declare at least one constructor, compilers won't generate default constructors, but because the classes fail to declare copy constructors or assignment operators, compilers will generate those functions (if they are needed).

Consider the following call to a copy constructor:

```
NamedObject<int> no1("Smallest Prime Number", 2);


NamedObject<int> no2(no1);      // calls copy constructor
```

The copy constructor generated by your compilers must initialize `no2.nameValue` and `no2.objectValue` using `no1.nameValue` and `no1.objectValue`, respectively. The type of `nameValue` is `string`, and `string` has a copy constructor (which you can verify by examining `string` in the standard library — see Item 49), so `no2.nameValue` will be initialized by calling the `string` copy constructor with `no1.nameValue` as its argument. On the other hand, the type of `NamedObject<int>::objectValue` is `int` (because `T` is `int` for this template instantiation), and no copy constructor is defined for `int`s, so `no2.objectValue` will be initialized by copying the bits over from `no1.objectValue`.

The compiler-generated assignment operator for `NamedObject<int>` would behave the same way, but in general, compiler-generated assignment operators behave as I've described only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will refuse to generate an `operator=` for your class, and you'll receive some lovely diagnostic during compilation.

For example, suppose `NamedObject` were defined like this, where `nameValue` is a *reference* to a string and `objectValue` is a *`const` `T`*:

```
template<class T>
class NamedObject {
public:
  // this ctor no longer takes a const name, because name-
  // Value is now a reference-to-non-const string. The char*
  // ctor is gone, because we must have a string to refer to
  NamedObject(string& name, const T& value);

  ...                           // as above, assume no
                                // operator= is declared
private:
  string& nameValue;            // this is now a reference
  const T objectValue;          // this is now const
};
```

Now consider what should happen here:

```
string newDog("Persephone");
string oldDog("Satch");

NamedObject<int> p(newDog, 2);      // as I write this, our dog
                                    // ◦Persephone is about to
                                    // have her second birthday

NamedObject<int> s(oldDog, 29);     // the family dog Satch
                                    // (from my childhood)
                                    // would be 29 if she were
                                    // still alive

p = s;                              // what should happen to
```

```
                                        // the data members in p?
```

Before the assignment, `p.nameValue` refers to some `string` object and `s.nameValue` also refers to a `string`, though not the same one. How should the assignment affect `p.nameValue`? After the assignment, should `p.nameValue` refer to the `string` referred to by `s.nameValue`, i.e., should the reference itself be modified? If so, that breaks new ground, because C++ doesn't provide a way to make a reference refer to a different object (see Item M1). Alternatively, should the `string` object to which `p.nameValue` refers be modified, thus affecting other objects that hold pointers or references to that `string`, i.e., objects not directly involved in the assignment? Is that what the compiler-generated assignment operator should do?

Faced with such a conundrum, C++ refuses to compile the code. If you want to support assignment in a class containing a reference member, you must define the assignment operator yourself. Compilers behave similarly for classes containing `const` members (such as `objectValue` in the modified class above); it's not legal to modify `const` members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers refuse to generate assignment operators for derived classes that inherit from base classes declaring the standard assignment operator `private`. After all, compiler-generated assignment operators for derived classes are supposed to handle base class parts, too (see Items 16 and M33), but in doing so, they certainly shouldn't invoke member functions the derived class has no right to call.

All this talk of compiler-generated functions gives rise to the question, what do you do if you want to disallow use of those functions? That is, what if you deliberately don't declare, for example, an `operator=` because you never *ever* want to allow assignment of objects in your class? The solution to that little teaser is the subject of Item 27. For a discussion of the often-overlooked interactions between pointer members and compiler-generated copy constructors and assignment operators, check out Item 11.

Back to Miscellany
Continue to Item 46: Prefer compile-time and link-time errors to runtime errors.

Back to Item 45: Know what functions C++ silently writes and calls.
Continue to Item 47: Ensure that non-local static objects are initialized before they're used.

Item 46: Prefer compile-time and link-time errors to runtime errors.

Other than in the few situations that cause C++ to throw exceptions (e.g., running out of memory — see Item 7), the notion of a runtime error is as foreign to C++ as it is to C. There's no detection of underflow, overflow, division by zero, no checking for array bounds violations, etc. Once your program gets past a compiler and linker, you're on your own — there's no safety net of any consequence. Much as with skydiving, some people are exhilarated by this state of affairs, others are paralyzed with fear. The motivation behind the philosophy, of course, is efficiency: without runtime checks, programs are smaller and faster.

There is a different way to approach things. Languages like Smalltalk and LISP generally detect fewer kinds of errors during compilation and linking, but they provide hefty runtime systems that catch errors during execution. Unlike C++, these languages are almost always interpreted, and you pay a performance penalty for the extra flexibility they offer.

Never forget that you are programming in C++. Even if you find the Smalltalk/LISP philosophy appealing, put it out of your mind. There's a lot to be said for adhering to the party line, and in this case, that means eschewing runtime errors. Whenever you can, push the detection of an error back from runtime to link-time, or, ideally, to compile-time.

Such a methodology pays dividends not only in terms of program size and speed, but also in terms of reliability. If your program gets through compilers and a linker without eliciting error messages, you may be confident there aren't any compiler- or linker-detectable errors in your program, period. (The other possibility, of course, is that there are bugs in your compilers or linkers, but let us not depress ourselves by admitting to such possibilities.)

With runtime errors, the situation is very different. Just because your program doesn't generate any runtime errors during a particular run, how can you be sure it won't generate errors during a different run, when you do things in a different order, use different data, or run for a longer or shorter period of time? You can test your program until you're blue in the face, but you'll still never cover all the possibilities. As a result, detecting errors at runtime is simply less secure than is catching them during compilation or linking.

Often, by making relatively minor changes to your design, you can catch during compilation what might otherwise be a runtime error. This frequently involves the addition of new types to the program. (See also Item M33.) For example, suppose you are writing a class to represent dates in time. Your first cut might look like this:

```
class Date {
```

```
public:
  Date(int day, int month, int year);


   ...


};
```

If you were to implement this constructor, one of the problems you'd face would be that of sanity checking on the values for the day and the month. Let's see how you can eliminate the need to validate the value passed in for the month.

One obvious approach is to employ an enumerated type instead of an integer:

```
enum Month { Jan = 1, Feb = 2, ... , Nov = 11, Dec = 12 };


class Date {
public:
  Date(int day, Month month, int year);


   ...


};
```

Unfortunately, this doesn't buy you that much, because enums don't have to be initialized:

```
Month m;
Date d(22, m, 1857);      // m is undefined
```

As a result, the `Date` constructor would still have to validate the value of the `month` parameter.

To achieve enough security to dispense with runtime checks, you've got to use a class to represent months, and you must ensure that only valid months are created:

```
class Month {
public:
  static const Month Jan() { return 1; }
```

```
  static const Month Feb() { return 2; }
  ...
  static const Month Dec() { return 12; }



  int asInt() const           // for convenience, make
  { return monthNumber; }     // it possible to convert
                              // a Month to an int


private:
  Month(int number): monthNumber(number) {}


  const int monthNumber;
};


class Date {
public:
  Date(int day, const Month& month, int year);
  ...
};
```

Several aspects of this design combine to make it work the way it does. First, the `Month` constructor is private. This prevents clients from creating new months. The only ones available are those returned by `Month`'s static member functions, plus copies thereof. Second, each `Month` object is `const`, so it can't be changed. (Otherwise the temptation to transform January into June might sometimes prove overwhelming, at least in northern latitudes.) Finally, the only way to get a `Month` object is by calling a function or by copying an existing `Month` (via the implicit `Month` copy constructor — see Item 45). This makes it possible to use `Month` objects anywhere and anytime; there's no need to worry about accidently using one before it's been initialized. (Item 47 explains why this might otherwise be a problem.)

Given these classes, it is all but impossible for a client to specify an invalid month. It would be completely impossible were it not for the following abomination:

```
Month *pm;                // define uninitialized ptr


Date d(1, *pm, 1997);     // arghhh! use it!
```

However, this involves dereferencing an uninitialized pointer, the results of which are undefined.

(See Item 3 for my feelings about undefined behavior.) Unfortunately, I know of no way to prevent or detect this kind of heresy. However, if we assume this never happens, or if we don't care how our software behaves if it does, the `Date` constructor can dispense with sanity checking on its `Month` parameter. On the other hand, the constructor must still check the `day` parameter for validity — how many days hath September, April, June, and November?

This `Date` example replaces runtime checks with compile-time checks. You may be wondering when it is possible to use link-time checks. In truth, not very often. C++ uses the linker to ensure that needed functions are defined exactly once (see Item 45 for a description of what it takes to "need" a function). It also uses the linker to ensure that static objects (see Item 47) are defined exactly once. You'll tend to use the linker in the same way. For example, Item 27 describes how the linker's checks can make it useful to deliberately avoid defining a function you explicitly declare.

Now don't get carried away. It's impractical to eliminate the need for *all* runtime checking. Any program that accepts interactive input, for example, is likely to have to validate that input. Similarly, a class implementing arrays that perform bounds checking (see Item 18) is usually going to have to validate the array index against the bounds every time an array access is made. Nonetheless, shifting checks from runtime to compile- or link-time is always a worthwhile goal, and you should pursue that goal whenever it is practical. Your reward for doing so is programs that are smaller, faster, and more reliable.

Back to Item 45: Know what functions C++ silently writes and calls.
Continue to Item 47: Ensure that non-local static objects are initialized before they're used.

Back to Item 46: Prefer compile-time and link-time errors to runtime errors.
Continue to Item 48: Pay attention to compiler warnings.

Item 47: Ensure that non-local static objects are initialized before they're used.

You're an adult now, so you don't need me to tell you it's foolhardy to use an object before it's been initialized. In fact, the whole notion may strike you as absurd; constructors make sure objects are initialized when they're created, *n'est-ce pas*?

Well, yes and no. Within a particular translation unit (i.e., source file), everything works fine, but things get trickier when the initialization of an object in one translation unit depends on the value of another object in a different translation unit *and* that second object itself requires initialization.

For example, suppose you've authored a library offering an abstraction of a file system, possibly including such capabilities as making files on the Internet look like they're local. Since your library makes the world look like a single file system, you might create a special object, `theFileSystem`, within your library's namespace (see Item 28) for clients to use whenever they need to interact with the file system abstraction your library provides:

```
class FileSystem { ... };           // this class is in your
                                    // library


FileSystem theFileSystem;           // this is the object
                                    // with which library
                                    // clients interact
```

Because `theFileSystem` represents something complicated, it's no surprise that its construction is both nontrivial and essential; use of `theFileSystem` before it had been constructed would yield *very* undefined behavior. (However, consult Item M17 for ideas on how the effective initialization of objects like `theFileSystem` can safely be delayed.)

Now suppose some client of your library creates a class for directories in a file system. Naturally, their class uses `theFileSystem`:

```
class Directory {                   // created by library client
public:
  Directory();
  ...
```

```
};


Directory::Directory()
{
  create a Directory object by invoking member
  functions on theFileSystem;
}
```

Further suppose this client decides to create a distinguished global `Directory` object for temporary files:

```
Directory tempDir;                  // directory for temporary
                                    // files
```

Now the problem of initialization order becomes apparent: unless `theFileSystem` is initialized before `tempDir`, `tempDir`'s constructor will attempt to use `theFileSystem` before it's been initialized. But `theFileSystem` and `tempDir` were created by different people at different times in different files. How can you be sure that `theFileSystem` will be created before `tempDir`?

This kind of question arises anytime you have *non-local static objects* that are defined in different translation units and whose correct behavior is dependent on their being initialized in a particular order. Non-local static objects are objects that are

- defined at global or namespace scope (e.g., `theFileSystem` and `tempDir`),
- declared `static` in a class, or
- defined `static` at file scope.

Regrettably, there is no shorthand term for "non-local static objects," so you should accustom yourself to this somewhat awkward phrase.

You do not want the behavior of your software to be dependent on the initialization order of non-local static objects in different translation units, because you have no control over that order. Let me repeat that. *You have absolutely no control over the order in which non-local static objects in different translation units are initialized.*

It is reasonable to wonder why this is the case.

It is the case because determining the "proper" order in which to initialize non-local static objects is hard. Very hard. Halting-Problem hard. In its most general form — with multiple translation units and non-local static objects generated through implicit template instantiations (which may

themselves arise via implicit template instantiations) — it's not only impossible to determine the right order of initialization, it's typically not even worth looking for special cases where it is possible to determine the right order.

In the field of Chaos Theory, there is a principle known as the "Butterfly Effect." This principle asserts that the tiny atmospheric disturbance caused by the beating of a butterfly's wings in one part of the world can lead to profound changes in weather patterns in places far distant. Somewhat more rigorously, it asserts that for some types of systems, minute perturbations in inputs can lead to radical changes in outputs.

The development of software systems can exhibit a Butterfly Effect of its own. Some systems are highly sensitive to the particulars of their requirements, and small changes in requirements can significantly affect the ease with which a system can be implemented. For example, Item 29 describes how changing the specification for an implicit conversion from `String`-to-`char*` to `String`-to-*`const`*-`char*` makes it possible to replace a slow or error-prone function with a fast, safe one.

The problem of ensuring that non-local static objects are initialized before use is similarly sensitive to the details of what you want to achieve. If, instead of demanding access to non-local static objects, you're willing to settle for access to objects that *act* like non-local static objects (except for the initialization headaches), the hard problem vanishes. In its stead is left a problem so easy to solve, it's hardly worth calling a problem any longer.

The technique — sometimes known as the *Singleton pattern* — is simplicity itself. First, you move each non-local static object into its own function, where you declare it `static`. Next, you have the function return a reference to the object it contains. Clients call the function instead of referring to the object. In other words, you replace non-local static objects with objects that are `static` inside functions. (See also Item M26.)

The basis of this approach is the observation that although C++ says next to nothing about when a non-local static object is initialized, it specifies quite precisely when a static object inside a function (i.e. a *local* static object) is initialized: it's when the object's definition is first encountered during a call to that function. So if you replace direct accesses to non-local static objects with calls to functions that return references to local static objects inside them, you're guaranteed that the references you get back from the functions will refer to initialized objects. As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can't be said for true non-local static objects.

Here's the technique applied to both `theFileSystem` and `tempDir`:

```
class FileSystem { ... };           // same as before
```

```
FileSystem& theFileSystem()          // this function replaces
{                                    // the theFileSystem object


  static FileSystem tfs;             // define and initialize
                                     // a local static object
                                     // (tfs = "the file system")


  return tfs;                        // return a reference to it
}


class Directory { ... };             // same as before


Directory::Directory()
{
  same as before, except references to theFileSystem are
  replaced by references to theFileSystem();
}


Directory& tempDir()                 // this function replaces
{                                    // the tempDir object


  static Directory td;               // define/initialize local
                                     // static object


  return td;                         // return reference to it
}
```

Clients of this modified system program exactly as they used to, except they now refer to `theFileSystem()` and `tempDir()` instead of `theFileSystem` and `tempDir`. That is, they refer only to functions returning references to those objects, never to the objects themselves.

The reference-returning functions dictated by this scheme are always simple: define and initialize a local static object on line 1, return it on line 2. That's it. Because they're so simple, you may be tempted to declare them `inline`. Item 33 explains that late-breaking revisions to the C++ language specification make this a perfectly valid implementation strategy, but it also explains why you'll want to confirm your compilers' conformance with this aspect of the standard before putting it to use. If you try it with a compiler not yet in accord with the relevant parts of the standard, you risk getting multiple copies of both the access function and the static object defined within it. That's enough to make a grown programmer cry.

Now, there's no magic going on here. For this technique to be effective, it must be possible to come up with a reasonable initialization order for your objects. If you set things up such that object A must be initialized before object B, and you also make A's initialization dependent on B's having already been initialized, you are going to get in trouble, and frankly, you deserve it. If you steer shy of such pathological situations, however, the scheme described in this Item should serve you quite nicely.

Back to Item 46: Prefer compile-time and link-time errors to runtime errors.
Continue to Item 48: Pay attention to compiler warnings.

Back to Item 47: Ensure that non-local static objects are initialized before they're used.
Continue to Item 49: Familiarize yourself with the standard library.

Item 48: Pay attention to compiler warnings.

Many programmers routinely ignore compiler warnings. After all, if the problem were serious, it'd be an error, right? This kind of thinking may be relatively harmless in other languages, but in C++, it's a good bet compiler writers have a better grasp of what's going on than you do. For example, here's an error everybody makes at one time or another:

```
class B {
public:
  virtual void f() const;
};

class D: public B {
public:
  virtual void f();
};
```

The idea is for `D::f` to redefine the virtual function `B::f`, but there's a mistake: in `B`, `f` is a `const` member function, but in `D` it's not declared `const`. One compiler I know says this about that:

```
warning: D::f() hides virtual B::f()
```

Too many inexperienced programmers respond to this message by saying to themselves, "Of *course* `D::f` hides `B::f` — that's what it's *supposed* to do!" Wrong. What this compiler is trying to tell you is that the `f` declared in `B` has not been redeclared in `D`, it's been hidden entirely (see Item 50 for a description of why this is so). Ignoring this compiler warning will almost certainly lead to erroneous program behavior, followed by a lot of debugging to find out about something that this compiler detected in the first place.

After you gain experience with the warning messages from a particular compiler, of course, you'll learn to understand what the different messages mean (which is often very different from what they *seem* to mean, alas). Once you have that experience, there may be a whole range of warnings you'll choose to ignore. That's fine, but it's important to make sure that before you dismiss a warning, you understand exactly what it's trying to tell you.

As long as we're on the topic of warnings, recall that warnings are inherently implementation-dependent, so it's not a good idea to get sloppy in your programming, relying on compilers to spot your mistakes for you. The function-hiding code above, for instance, goes through a different (but widely used) compiler with nary a squawk. Compilers are supposed to translate C++ into an

executable format, not act as your personal safety net. You want that kind of safety? Program in Ada.

Back to Item 47: Ensure that non-local static objects are initialized before they're used.
Continue to Item 49: Familiarize yourself with the standard library.

Back to Item 48: Pay attention to compiler warnings.
Continue to Item 50: Improve your understanding of C++.

Item 49: Familiarize yourself with the standard library.

C++'s standard library is big. Very big. Incredibly big. How big? Let me put it this way: the specification takes over 300 closely-packed pages in the ◦C++ standard, and that all but excludes the standard C library, which is included in the C++ library "by reference." (That's the term they use, honest.)

Bigger isn't always better, of course, but in this case, bigger *is* better, because a big library contains lots of functionality. The more functionality in the standard library, the more functionality you can lean on as you develop your applications. The C++ library doesn't offer *everything* (support for concurrency and for graphical user interfaces is notably absent), but it does offer a lot. You can lean almost anything against it.

Before summarizing what's in the library, I need to tell you a bit about how it's organized. Because the library has so much in it, there's a reasonable chance you (or someone like you) may choose a class or function name that's the same as a name in the standard library. To shield you from the name conflicts that would result, virtually everything in the standard library is nestled in the namespace `std` (see Item 28). But that leads to a new problem. Gazillions of lines of existing C++ rely on functionality in the pseudo-standard library that's been in use for years, e.g., functionality declared in the headers `<iostream.h>`, `<complex.h>`, `<limits.h>`, etc. That existing software isn't designed to use namespaces, and it would be a shame if wrapping the standard library by `std` caused the existing code to break. (Authors of the broken code would likely use somewhat harsher language than "shame" to describe their feelings about having the library rug pulled out from underneath them.)

Mindful of the destructive power of rioting bands of incensed programmers, the ◦standardization committee decided to create new header names for the `std`-wrapped components. The algorithm they chose for generating the new header names is as trivial as the results it produces are jarring: the `.h` on the existing C++ headers was simply dropped. So `<iostream.h>` became `<iostream>`, `<complex.h>` became `<complex>`, etc. For C headers, the same algorithm was applied, but a `c` was prepended to each result. Hence C's `<string.h>` became `<cstring>`, `<stdio.h>` became `<cstdio>`, etc. For a final twist, the old C++ headers were officially *deprecated* (i.e., listed as no longer supported), but the old C headers were not (to maintain C compatibility). In practice, compiler vendors have no incentive to disavow their customers' legacy software, so you can expect the old C++ headers to be supported for many years.

Practically speaking, then, this is the C++ header situation:

- Old C++ header names like `<iostream.h>` are likely to continue to be supported, even though they aren't in the ◦official standard. The contents of such headers are *not* in namespace `std`.

- New C++ header names like `<iostream>` contain the same basic functionality as the corresponding old headers, but the contents of the headers *are* in namespace `std`. (During standardization, the details of some of the library components were modified, so there isn't necessarily an exact match between the entities in an old C++ header and those in a new one.)
- Standard C headers like `<stdio.h>` continue to be supported. The contents of such headers are *not* in `std`.
- New C++ headers for the functionality in the C library have names like `<cstdio>`. They offer the same contents as the corresponding old C headers, but the contents *are* in `std`.

All this seems a little weird at first, but it's really not that hard to get used to. The biggest challenge is keeping all the string headers straight: `<string.h>` is the old C header for `char*`-based string manipulation functions, `<string>` is the `std`-wrapped C++ header for the new string classes (see below), and `<cstring>` is the `std`-wrapped version of the old C header. If you can master that (and I know you can), the rest of the library is easy.

The next thing you need to know about the standard library is that almost everything in it is a template. Consider your old friend iostreams. (If you and iostreams aren't friends, turn to Item 2 to find out why you should cultivate a relationship.) Iostreams help you manipulate streams of characters, but what's a character? Is it a `char`? A `wchar_t`? A Unicode character? Some other multi-byte character? There's no obviously right answer, so the library lets you choose. All the stream classes are really class templates, and you specify the character type when you instantiate a stream class. For example, the standard library defines the type of `cout` to be `ostream`, but `ostream` is really a typedef for `basic_ostream<char>`.

Similar considerations apply to most of the other classes in the standard library. `string` isn't a class, it's a class template: a type parameter defines the type of characters in each `string` class. `complex` isn't a class, it's a class template: a type parameter defines the type of the real and imaginary components in each `complex` class. `vector` isn't a class, it's a class template. On and on it goes.

You can't escape the templates in the standard library, but if you're used to working with only streams and strings of `chars`, you can mostly ignore them. That's because the library defines typedefs for `char` instantiations for these components of the library, thus letting you continue to program in terms of the objects `cin`, `cout`, `cerr`, etc., and the types `istream`, `ostream`, `string`, etc., without having to worry about the fact that `cin`'s real type is `basic_istream<char>` and `string`'s is `basic_string<char>`.

Many components in the standard library are templatized much more than this suggests. Consider again the seemingly straightforward notion of a string. Sure, it can be parameterized based on the type of characters it holds, but different character sets differ in details, e.g., special end-of-file characters, most efficient way of copying arrays of them, etc. Such characteristics are known in the standard as *traits*, and they are specified for `string` instantiations by an additional template parameter. In addition, `string` objects are likely to perform dynamic memory allocation and deallocation, but there are lots of different ways to approach that task (see Item 10). Which is best?

You get to choose: the `string` template takes an `Allocator` parameter, and objects of type `Allocator` are used to allocate and deallocate the memory used by `string` objects.

Here's a full-blown declaration for the `basic_string` template and the `string` typedef that builds on it; you can find this (or something equivalent to it) in the header `<string>`:

```
namespace std {


  template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
     class basic_string;


  typedef basic_string<char> string;


}
```

Notice how `basic_string` has default values for its `traits` and `Allocator` parameters. This is typical of the standard library. It offers flexibility to those who need it, but "typical" clients who just want to do the "normal" thing can ignore the complexity that makes possible the flexibility. In other words, if you just want string objects that act more or less like C strings, you can use `string` objects and remain merrily ignorant of the fact that you're really using objects of type `basic_string<char, char_traits<char>, allocator<char> >`.

Well, usually you can. Sometimes you have to peek under the hood a bit. For example, Item 34 discusses the advantages of declaring a class without providing its definition, and it remarks that the following is the wrong way to declare the `string` type:

```
class string;                   // this will compile, but
                                // you don't want to do it
```

Setting aside namespace considerations for a moment, the real problem here is that `string` isn't a class, it's a typedef. It would be nice if you could solve the problem this way:

```
typedef basic_string<char> string;
```

but that won't compile. "What is this `basic_string` of which you speak?," your compilers will

wonder, though they'll probably phrase the question rather differently. No, to declare `string`, you would first have to declare all the templates on which it depends. If you could do it, it would look something like this:

```
template<class charT> struct char_traits;


template<class T> class allocator;


  template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
     class basic_string;


typedef basic_string<char> string;
```

However, you can't declare `string`. At least you shouldn't. That's because library implementers are allowed to declare `string` (or anything else in the `std` namespace) differently from what's specified in the standard as long as the result offers standard-conforming behavior. For example, a `basic_string` implementation could add a fourth template parameter, but that parameter's default value would have to yield code that acts as the standard says an unadorned `basic_string` must.

End result? Don't try to manually declare `string` (or any other part of the standard library). Instead, just include the appropriate header, e.g. `<string>`.

With this background on headers and templates under our belts, we're in a position to survey the primary components of the standard C++ library:

- **The standard C library.** It's still there, and you can still use it. A few minor things have been tweaked here and there, but for all intents and purposes, it's the same C library that's been around for years.
- **Iostreams.** Compared to "traditional" iostream implementations, it's been templatized, its inheritance hierarchy has been modified, it's been augmented with the ability to throw exceptions, and it's been updated to support strings (via the `stringstream` classes) and internationalization (via locales — see below). Still, most everything you've come to expect from the iostream library continues to exist. In particular, it still supports stream buffers, formatters, manipulators, and files, plus the objects `cin`, `cout`, `cerr`, and `clog`. That means you can treat `string`s and files as streams, and you have extensive control over stream behavior, including buffering and formatting.
- **Strings.** `string` objects were designed to eliminate the need to use `char*` pointers in most applications. They support the operations you'd expect (e.g., concatenation, constant-time

access to individual characters via `operator[]`, etc.), they're convertible to `char*`s for compatibility with legacy code, and they handle memory management automatically. Some `string` implementations employ reference counting (see Item M29), which can lead to *better* performance (in both time and space) than `char*`-based strings.

- **Containers.** Stop writing your own basic container classes! The library offers efficient implementations of vectors (they act like dynamically extensible arrays), lists (doubly-linked), queues, stacks, deques, maps, sets, and bitsets. Alas, there are no hash tables in the library (though many vendors offer them as extensions), but compensating somewhat is the fact that `string`s are containers. That's important, because it means anything you can do to a container (see below), you can also do to a `string`.

  What's that? You want to know how I *know* the library implementations are efficient? Easy: the library specifies each class's interface, and part of each interface specification is a set of performance guarantees. So, for example, no matter how `vector` is implemented, it's not enough to offer just *access* to its elements, it must offer *constant-time* access. If it doesn't, it's not a valid `vector` implementation.

  In many C++ programs, dynamically allocated strings and arrays account for most uses of `new` and `delete`, and `new`/`delete` errors — especially leaks caused by failure to delete `new`ed memory — are distressingly common. If you use `string` and `vector` objects (both of which perform their own memory management) instead of `char*`s and pointers to dynamically allocated arrays, many of your `new`s and `delete`s will vanish, and so will the difficulties that frequently accompany their use (e.g., Items 6 and 11).
- **Algorithms.** Having standard containers is nice, but it's even nicer when there's an easy way to do things with them. The standard library offers over two dozen easy ways (i.e., predefined functions, officially known as *algorithms* — they're really function templates), most of which work with *all* the containers in the library — as well as with built-in arrays!

  Algorithms treat the contents of a container as a sequence, and each algorithm may be applied to either the sequence corresponding to all the values in a container or to a subsequence. Among the standard algorithms are `for_each` (apply a function to each element of a sequence), `find` (find the first location in a sequence holding a given value — Item M35 shows its implementation), `count_if` (count the number of elements in a sequence for which a given predicate is true), `equal` (determine whether two sequences hold equal-valued elements), `search` (find the first position in one sequence where a second sequence occurs as a subsequence), `copy` (copy one sequence into another), `unique` (remove duplicate values from a sequence), `rotate` (rotate the values in a sequence) and `sort` (sort the values in a sequence). Note that this is just a *sampling* of the algorithms available; the library contains many others.

  Just as container operations come with performance guarantees, so do algorithms. For example, the `stable_sort` algorithm is required to perform no more than O(N log N) comparisons. (If the "Big O" notation in the previous sentence is foreign to you, don't sweat it. What it really means is that, broadly speaking, `stable_sort` must offer performance at the

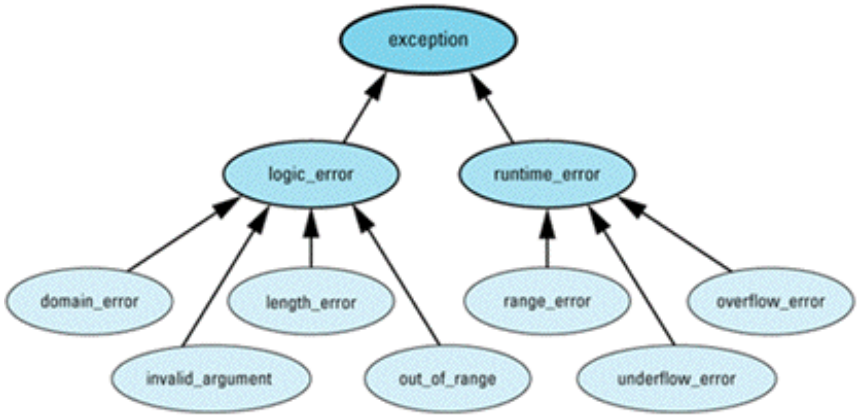
same level as the most efficient general-purpose serial sorting algorithms.)

- **Support for internationalization.** Different cultures do things in different ways. Like the C library, the C++ library offers features to facilitate the production of internationalized software, but the C++ approach, though conceptually akin to that of C, is different. It should not surprise you, for example, to learn that C++'s support for internationalization makes extensive use of templates, and it takes advantage of inheritance and virtual functions, too.

  The primary library components supporting internationalization are *facets* and *locales*. Facets describe how particular characteristics of a culture should be handled, including collation rules (i.e., how strings in the local character set should be sorted), how dates and times should be expressed, how numeric and monetary values should be presented, how to map from message identifiers to (natural) language-specific messages, etc. Locales bundle together sets of facets. For example, a locale for the United States would include facets describing how to sort strings in American English, read and write dates and times, read and write monetary and numeric values, etc., in a way appropriate for people in the USA. A locale for France, on the other hand, would describe how to perform these tasks in a manner to which the French are accustomed. C++ allows multiple locales to be active within a single program, so different parts of an application may employ different conventions.

- **Support for numeric processing.** The end for FORTRAN may finally be near. The C++ library offers a template for complex number classes (the precision of the real and imaginary parts may be `float`, `double`, or `long double`) as well as for special array types specifically designed to facilitate numeric programming. Objects of type `valarray`, for example, are defined to hold elements that are free from aliasing. This allows compilers to be much more aggressive in their optimizations, especially for vector machines. The library also offers support for two different types of array slices, as well as providing algorithms to compute inner products, partial sums, adjacent differences, and more.

- **Diagnostic support.** The standard library offers support for three ways to report errors: via C's assertions (see Item 7), via error numbers, and via exceptions. To help provide some structure to exception types, the library defines the following hierarchy of exception classes:



Exceptions of type `logic_error` (or its subclasses) represent errors in the logic of software. In theory, such errors could have been prevented by more careful programming. Exceptions of type `runtime_error` (or its derived classes) represent errors detectable only at runtime.

You may use these classes as is, you may inherit from them to create your own exception classes, or you may ignore them. Their use is not mandatory.

This list doesn't describe everything in the standard library. Remember, the specification runs over 300 pages. Still, it should give you the basic lay of the land.

The part of the library pertaining to containers and algorithms is commonly known as *Standard Template Library* (the STL — see Item M35). There is actually a third component to the STL — Iterators — that I haven't described. Iterators are pointer-like objects that allow STL algorithms and containers to work together. You need not understand iterators for the high-level description of the standard library I give here. If you're interested in them, however, you can find examples of their use in Items 39 and M35.

The STL is the most revolutionary part of the standard library, not because of the containers and algorithms it offers (though they are undeniably useful), but because of its architecture. Simply put, the architecture is extensible: you can *add* to the STL. Of course, the components of the standard library itself are fixed, but if you follow the conventions on which the STL is built, you can write your own containers, algorithms, and iterators that work as well with the standard STL components as the STL components work with one another. You can also take advantage of STL-compliant containers, algorithms, and iterators written by others, just as they can take advantage of yours.

What makes the STL revolutionary is that it's not really software, it's a set of *conventions*. The STL components in the standard library are simply manifestations of the good that can come from following those conventions.

By using the components in the standard library, you can generally dispense with designing your own from-the-ground-up mechanisms for stream I/O, strings, containers (including iteration and common manipulations), internationalization, numeric data structures, and diagnostics. That leaves you a lot more time and energy for the really important part of software development: implementing the things that distinguish your wares from those of your competitors.

Back to Item 48: Pay attention to compiler warnings.
Continue to Item 50: Improve your understanding of C++.

Back to Item 49: Familiarize yourself with the standard library.
Continue to Afterword

Item 50: Improve your understanding of C++.

There's a lot of *stuff* in C++. C stuff. Overloading stuff. Object-oriented stuff. Template stuff. Exception stuff. Namespace stuff. Stuff, stuff, stuff! Sometimes it can be overwhelming. How do you make sense of all that stuff?

It's not that hard once you understand the design goals that forged C++ into what it is. Foremost amongst those goals are the following:

- **Compatibility with C.** Lots and lots of C exists, as do lots and lots of C programmers. C++ takes advantage of and builds on — er, I mean it "leverages" — that base.
- **Efficiency.** ◦Bjarne Stroustrup, the designer and first implementer of C++, knew from the outset that the C programmers he hoped to win over wouldn't look twice if they had to pay a performance penalty for switching languages. As a result, he made sure C++ was competitive with C when it came to efficiency — like within 5%.
- **Compatibility with traditional tools and environments.** Fancy development environments run here and there, but compilers, linkers, and editors run almost everywhere. C++ is designed to work in environments from mice to mainframes, so it brings along as little baggage as possible. You want to port C++? You port a *language* and take advantage of existing tools on the target platform. (However, it is often possible to provide a better implementation if, for example, the linker can be modified to address some of the more demanding aspects of inlining and templates.)
- **Applicability to real problems.** C++ wasn't designed to be a nice, pure language, good for teaching students how to program, it was designed to be a powerful tool for professional programmers solving real problems in diverse domains. The real world has some rough edges, so it's no surprise there's the occasional scratch marring the finish of the tools on which the pros rely.

These goals explain a multitude of language details that might otherwise merely chafe. Why do implicitly-generated copy constructors and assignment operators behave the way they do, especially for pointers (see Items 11 and 45)? Because that's how C copies and assigns `structs`, and compatibility with C is important. Why aren't destructors automatically virtual (see Item 14), and why must implementation details appear in class definitions (see Item 34)? Because doing otherwise would impose a performance penalty, and efficiency is important. Why can't C++ detect initialization dependencies between non-local static objects (see Item 47)? Because C++ supports separate translation (i.e., the ability to compile source modules separately, then link several object files together to form an executable), relies on existing linkers, and doesn't mandate the existence of program databases. As a result, C++ compilers almost never know everything about an entire program. Finally, why doesn't C++ free programmers from tiresome duties like memory management (see Items 5-10) and low-level pointer manipulations? Because some programmers need those capabilities, and the needs of real programmers are of paramount importance.

This barely hints at how the design goals behind C++ shape the behavior of the language. To cover everything would take an entire book, so it's convenient that Stroustrup wrote one. That book is ◦*The Design and Evolution of C++* (Addison-Wesley, 1994), sometimes known as simply "D&E." Read it, and you'll see what features were added to C++, in what order, and why. You'll also learn about features that were rejected, and why. You'll even get the inside story on how the `dynamic_cast` feature (see Items 39 and M2) was considered, rejected, reconsidered, then accepted — and why. If you're having trouble making sense of C++, D&E should dispel much of your confusion.

The Design and Evolution of C++ offers a wealth of insights into how C++ came to be what it is, but it's nothing like a formal specification for the language. For that you must turn to the ◦international standard for C++, an impressive exercise in formalese running some 700 pages. There you can read such riveting prose as this:

> A virtual function call uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides.

This paragraph is the basis for Item 38 ("Never redefine an inherited default parameter value"), but I hope my treatment of the topic is somewhat more accessible than the text above.

The standard is hardly bedtime reading, but it's your best recourse — your *standard* recourse — if you and someone else (a compiler vendor, say, or a developer of some other tool that processes source code) disagree on what is and isn't C++. The whole purpose of a standard is to provide definitive information that settles arguments like that.

The standard's official title is a mouthful, but if you need to know it, you need to know it. Here it is: *International Standard for Information Systems—Programming Language C++*. It's published by Working Group 21 of the ◦International Organization for Standardization (ISO). (If you insist on being picky about it, it's really published by — I am not making this up — ISO/IEC JTC1/SC22/WG21.) You can order a copy of the official standard from your national standards body (in the United States, that's ANSI, the ◦American National Standards Institute), but copies of late drafts of the standard — which are quite similar (though not identical) to the final document — are freely available on the World Wide Web. A good place to look for a copy is at ◦the Cygnus Solutions Draft Standard C++ Page, but given the pace of change in cyberspace, don't be surprised if this link is broken by the time you try it. If it is, your favorite Web search engine will doubtless turn up a URL that works.

As I said, *The Design and Evolution of C++* is fine for insights into the language's design, and the standard is great for nailing down language details, but it would be nice if there were a comfortable

middle ground between D&E's view from 10,000 meters and the standard's micron-level examination. Textbooks are supposed to fill this niche, but they generally drift toward the standard's perspective, whereby *what* the language is receives a lot more attention than *why* it's that way.

Enter the ARM. The ARM is another book, ◦*The Annotated C++ Reference Manual*, by Margaret Ellis and ◦Bjarne Stroustrup (Addison-Wesley, 1990). Upon its publication, it became *the* authority on C++, and the international standard started with the ARM (along with the existing C standard) as its basis. In the intervening years, the language specified by the standard has in some ways parted company with that described by the ARM, so the ARM is no longer the authority it once was. It's still a useful reference, however, because most of what it says is still true, and it's not uncommon for vendors to adhere to the ARM specification in areas of C++ where the standard has only recently settled down.

What makes the ARM really useful, however, isn't the RM part (the Reference Manual), it's the A part: the annotations. The ARM provides extensive commentary on *why* many features of C++ behave the way they do. Some of this information is in D&E, but much of it isn't, and you *do* want to know it. For instance, here's something that drives most people crazy when they first encounter it:

```
class Base {
public:
  virtual void f(int x);
};

class Derived: public Base {
public:
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                              // error!
```

The problem is that `Derived::f` hides `Base::f`, even though they take different parameter types, so compilers demand that the call to `f` take a `double*`, which the literal `10` most certainly is not.

This is inconvenient, but the ARM provides an explanation for this behavior. Suppose that when you called `f`, you really did want to call the version in `Derived`, but you accidentally used the wrong parameter type. Further suppose that `Derived` is way down in an inheritance hierarchy and that you were unaware that `Derived` indirectly inherits from some base class `BaseClass`, and that `BaseClass` declares a virtual function `f` that takes an `int`. In that case, you would have inadvertently called `BaseClass::f`, a function you didn't even know existed! This kind of error could occur frequently where large class hierarchies are used, so Stroustrup decided to nip it in the bud by having derived class members hide base class members on a per-name basis.

Note, by the way, that if the writer of `Derived` wants to allow clients to access `Base::f`, this is easily accomplished via a simple `using` declaration:

```
class Derived: public Base {
public:
  using Base::f;                  // import Base::f into
                                  // Derived's scope
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                        // fine, calls Base::f
```

For compilers not yet supporting `using` declarations, an alternative is to employ an inline function:

```
class Derived: public Base {
public:
  virtual void f(int x) { Base::f(x); }
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                 // fine, calls Derived::f(int),
                           // which calls Base::f(int)
```

Between D&E and the ARM, you'll gain insights into the design and implementation of C++ that make it possible to appreciate the sound, no-nonsense architecture behind a sometimes baroque-looking facade. Fortify those insights with the detailed information in the standard, and you've got a foundation for software development that leads to truly *effective* C++.

Back to Item 49: Familiarize yourself with the standard library.
Continue to Afterword

Back to Item 50: Improve your understanding of C++.

Afterword

If, having digested 50 ways to improve your programs and designs, you still find yourself hungry for C++ guidelines, you may be interested in my second book on the subject, *More Effective C++: 35 New Ways to Improve Your Programs and Design*. Like *Effective C++*, More Effective C++ covers material that's essential for effective C++ software development, but *Effective C++* focuses more on fundamentals, while *More Effective C++* also spends time on newer language features and on advanced programming techniques.

You can find detailed information on *More Effective C++* — including four complete Items, the book's list of recommended reading, and more — at the *More Effective C++* web site. In case you can't wait, the contents of *More Effective C++* are summarized below.

Back to Item 50: Improve your understanding of C++.

More Effective C++

Contents

Dedication

For Clancy, my favorite enemy within.

Continue to Acknowledgements

Back to Dedication
Continue to Introduction

Acknowledgments

A great number of people helped bring this book into existence. Some contributed ideas for technical topics, some helped with the process of producing the book, and some just made life more fun while I was working on it.

When the number of contributors to a book is large, it is not uncommon to dispense with individual acknowledgments in favor of a generic "Contributors to this book are too numerous to mention." I prefer to follow the expansive lead of John L. Hennessy and David A. Patterson in ◦*Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, 1995). In addition to motivating the comprehensive acknowledgments that follow, their book provides hard data for the 90-10 rule, which I refer to in Item 16.

The Items

With the exception of direct quotations, all the words in this book are mine. However, many of the ideas I discuss came from others. I have done my best to keep track of who contributed what, but I know I have included information from sources I now fail to recall, foremost among them many posters to the Usenet newsgroups ◦`comp.lang.c++` and ◦comp.std.c++.

Many ideas in the C++ community have been developed independently by many people. In what follows, I note only where *I* was exposed to particular ideas, not necessarily where those ideas originated.

Brian Kernighan suggested the use of macros to approximate the syntax of the new C++ casting operators I describe in Item 2.

In Item 3, my warning about deleting an array of derived class objects through a base class pointer is based on material in Dan Saks' "Gotchas" talk, which he's given at several conferences and trade shows.

In Item 5, the proxy class technique for preventing unwanted application of single-argument constructors is based on material in Andrew Koenig's column in the January 1994 ◦*C++ Report*.

James Kanze made a posting to ◦`comp.lang.c++` on implementing postfix increment and decrement operators via the corresponding prefix functions; I use his technique in Item 6.

David Cok, writing me about material I covered in *Effective C++*, brought to my attention the

distinction between `operator new` and the `new` operator that is the crux of Item 8. Even after reading his letter, I didn't really understand the distinction, but without his initial prodding, I probably *still* wouldn't.

The notion of using destructors to prevent resource leaks (used in Item 9) comes from section 15.3 of Margaret A. Ellis' and Bjarne Stroustrup's *The Annotated C++ Reference Manual* (see page 285). There the technique is called *resource acquisition is initialization*. Tom Cargill suggested I shift the focus of the approach from resource acquisition to resource release.

Some of my discussion in Item 11 was inspired by material in Chapter 4 of *Taligent's Guide to Designing Programs* (Addison-Wesley, 1994).

My description of over-eager memory allocation for the `DynArray` class in Item 18 is based on Tom Cargill's article, "A Dynamic vector is harder than it looks," in the June 1992 *C++ Report*. A more sophisticated design for a dynamic array class can be found in Cargill's follow-up column in the January 1994 *C++ Report*.

Item 21 was inspired by Brian Kernighan's paper, "An AWK to C++ Translator," at the 1991 USENIX C++ Conference. His use of overloaded operators (sixty-seven of them!) to handle mixed-type arithmetic operations, though designed to solve a problem unrelated to the one I explore in Item 21, led me to consider multiple overloadings as a solution to the problem of temporary creation.

In Item 26, my design of a template class for counting objects is based on a posting to `comp.lang.c++` by Jamshid Afshar.

The idea of a mixin class to keep track of pointers from `operator new` (see Item 27) is based on a suggestion by Don Box. Steve Clamage made the idea practical by explaining how `dynamic_cast` can be used to find the beginning of memory for an object.

The discussion of smart pointers in Item 28 is based in part on Steven Buroff's and Rob Murray's *C++ Oracle* column in the October 1993 *C++ Report*; on Daniel R. Edelson's classic paper, "Smart Pointers: They're Smart, but They're Not Pointers," in the proceedings of the 1992 USENIX C++ Conference; on section 15.9.1 of Bjarne Stroustrup's *The Design and Evolution of C++* (see page 285); on Gregory Colvin's "C++ Memory Management" class notes from C/C++ Solutions '95; and on Cay Horstmann's column in the March-April 1993 issue of the *C++ Report*. I developed some of the material myself, though. Really.

In Item 29, the use of a base class to store reference counts and of smart pointers to manipulate those counts is based on Rob Murray's discussions of the same topics in sections 6.3.2 and 7.4.2, respectively, of his *C++ Strategies and Tactics* (see page 286). The design for adding reference

counting to existing classes follows that presented by Cay Horstmann in his March-April 1993 column in the ◦*C++ Report*.

In Item 30, my discussion of lvalue contexts is based on comments in Dan Saks' column in the C User's Journal ◦C/C++ Users Journal) of January 1993. The observation that non-proxy member functions are unavailable when called through proxies comes from an unpublished paper by Cay Horstmann.

The use of runtime type information to build vtbl-like arrays of function pointers (in Item 31) is based on ideas put forward by Bjarne Stroustrup in postings to ◦`comp.lang.c++` and in section 13.8.1 of his ◦The Design and Evolution of C++ (see page 285).

The material in Item 33 is based on several of my ◦C++ Report columns in 1994 and 1995. Those columns, in turn, included comments I received from Klaus Kreft about how to use `dynamic_cast` to implement a virtual `operator=` that detects arguments of the wrong type.

Much of the material in Item 34 was motivated by Steve Clamage's article, "Linking C++ with other languages," in the May 1992 ◦C++ Report. In that same Item, my treatment of the problems caused by functions like `strdup` was motivated by an anonymous reviewer.

The Book

Reviewing draft copies of a book is hard — and vitally important — work. I am grateful that so many people were willing to invest their time and energy on my behalf. I am especially grateful to Jill Huchital, Tim Johnson, Brian Kernighan, Eric Nagler, and Chris Van Wyk, as they read the book (or large portions of it) more than once. In addition to these gluttons for punishment, complete drafts of the manuscript were read by Katrina Avery, Don Box, Steve Burkett, Tom Cargill, Tony Davis, Carolyn Duby, Bruce Eckel, Read Fleming, Cay Horstmann, James Kanze, Russ Paielli, Steve Rosenthal, Robin Rowe, Dan Saks, Chris Sells, Webb Stacy, Dave Swift, Steve Vinoski, and Fred Wild. Partial drafts were reviewed by Bob Beauchaine, Gerd Hoeren, Jeff Jackson, and Nancy L. Urbano. Each of these reviewers made comments that greatly improved the accuracy, utility, and presentation of the material you find here.

Once the book came out, I received corrections and suggestions from many people. I've listed these sharp-eyed readers in the order in which I received their missives: Luis Kida, John Potter, Tim Uttormark, Mike Fulkerson, Dan Saks, Wolfgang Glunz, Clovis Tondo, Michael Loftus, Liz Hanks, Wil Evers, Stefan Kuhlins, Jim McCracken, Alan Duchan, John Jacobsma, Ramesh Nagabushnam, Ed Willink, Kirk Swenson, Jack Reeves, Doug Schmidt, Tim Buchowski, Paul Chisholm, Andrew Klein, Eric Nagler, Jeffrey Smith, Sam Bent, Oleg Shteynbuk, Anton Doblmaier, Ulf Michaelis, Sekhar Muddana, Michael Baker, Yechiel Kimchi, David Papurt, Ian Haggard, Robert Schwartz, David Halpin, Graham Mark, David Barrett, Damian Kanarek, Ron Coutts, Lance Whitesel, Jon Lachelt, Cheryl Ferguson, Munir Mahmood, Klaus-Georg Adams, David Goh, Chris Morley, and Rainer Baumschlager. Their suggestions allowed me to improve *More Effective C++* in updated

printings (such as this one), and I greatly appreciate their help.

During preparation of this book, I faced many questions about the emerging ISO/ANSI standard for C++, and I am grateful to Steve Clamage and Dan Saks for taking the time to respond to my incessant email queries.

John Max Skaller and Steve Rumsby conspired to get me the HTML for the draft ANSI C++ standard before it was widely available. Vivian Neou pointed me to the Netscape WWW browser as a stand-alone HTML viewer under (16 bit) Microsoft Windows, and I am deeply grateful to the folks at Netscape Communications for making their fine viewer freely available on such a pathetic excuse for an operating system.

Bryan Hobbs and Hachemi Zenad generously arranged to get me a copy of the internal engineering version of the MetaWare C++ compiler so I could check the code in this book using the latest features of the language. Cay Horstmann helped me get the compiler up and running in the very foreign world of DOS and DOS extenders. Borland (now Inprise) provided a beta copy of their most advanced compiler, and Eric Nagler and Chris Sells provided invaluable help in testing code for me on compilers to which I had no access.

Without the staff at the Corporate and Professional Publishing Division of Addison-Wesley, there would be no book, and I am indebted to Kim Dawley, Lana Langlois, Simone Payment, Marty Rabinowitz, Pradeepa Siva, John Wait, and the rest of the staff for their encouragement, patience, and help with the production of this work.

Chris Guzikowski helped draft the back cover copy for this book, and Tim Johnson stole time from his research on low-temperature physics to critique later versions of that text.

Tom Cargill graciously agreed to make his *C++ Report* article on exceptions available.

The People

Kathy Reed was responsible for my introduction to programming; surely she didn't deserve to have to put up with a kid like me. Donald French had faith in my ability to develop and present C++ teaching materials when I had no track record. He also introduced me to John Wait, my editor at Addison-Wesley, an act for which I will always be grateful. The triumvirate at Beaver Ridge — Jayni Besaw, Lorri Fields, and Beth McKee — provided untold entertainment on my breaks as I worked on the book.

My wife, Nancy L. Urbano, put up with me and put up with me and put up with me as I worked on the book, continued to work on the book, and kept working on the book. How many times did she hear me say we'd do something after the book was done? Now the book is done, and we will do those things. She amazes me. I love her.

Finally, I must acknowledge our puppy, ◦Persephone, whose existence changed our world forever. Without her, this book would have been finished both sooner and with less sleep deprivation, but also with substantially less comic relief.

Back to Dedication
Continue to Introduction

Back to Acknowledgments
Continue to Basics

Introduction

These are heady days for C++ programmers. Commercially available less than a decade, C++ has nevertheless emerged as the language of choice for systems programming on nearly all major computing platforms. Companies and individuals with challenging programming problems increasingly embrace the language, and the question faced by those who do not use C++ is often *when* they will start, not *if*. Standardization of C++ is complete, and the breadth and scope of the accompanying library — which both dwarfs and subsumes that of C — makes it possible to write rich, complex programs without sacrificing portability or implementing common algorithms and data structures from scratch. C++ compilers continue to proliferate, the features they offer continue to expand, and the quality of the code they generate continues to improve. Tools and environments for C++ development grow ever more abundant, powerful, and robust. Commercial libraries all but obviate the need to write code in many application areas.

As the language has matured and our experience with it has increased, our needs for information about it have changed. In 1990, people wanted to know *what* C++ was. By 1992, they wanted to know *how* to make it work. Now C++ programmers ask higher-level questions: How can I design my software so it will adapt to future demands? How can I improve the efficiency of my code without compromising its correctness or making it harder to use? How can I implement sophisticated functionality not directly supported by the language?

In this book, I answer these questions and many others like them.

This book shows how to design and implement C++ software that is *more effective*: more likely to behave correctly; more robust in the face of exceptions; more efficient; more portable; makes better use of language features; adapts to change more gracefully; works better in a mixed-language environment; is easier to use correctly; is harder to use incorrectly. In short, software that's just *better*.

The material in this book is divided into 35 Items. Each Item summarizes accumulated wisdom of the C++ programming community on a particular topic. Most Items take the form of guidelines, and the explanation accompanying each guideline describes why the guideline exists, what happens if you fail to follow it, and under what conditions it may make sense to violate the guideline anyway.

Items fall into several categories. Some concern particular language features, especially newer features with which you may have little experience. For example, Items 9 through 15 are devoted to exceptions (as are the magazine articles by Tom Cargill, Jack Reeves, and Herb Sutter). Other Items explain how to combine the features of the language to achieve higher-level goals. Items 25

through 31, for instance, describe how to constrain the number or placement of objects, how to create functions that act "virtual" on the type of more than one object, how to create "smart pointers," and more. Still other Items address broader topics; Items 16 through 24 focus on efficiency. No matter what the topic of a particular Item, each takes a no-nonsense approach to the subject. In *More Effective C++*, you learn how to *use* C++ more effectively. The descriptions of language features that make up the bulk of most C++ texts are in this book mere background information.

An implication of this approach is that you should be familiar with C++ before reading this book. I take for granted that you understand classes, protection levels, virtual and nonvirtual functions, etc., and I assume you are acquainted with the concepts behind templates and exceptions. At the same time, I don't expect you to be a language expert, so when poking into lesser-known corners of C++, I always explain what's going on.

The C++ in *More Effective C++*

The C++ I describe in this book is the language specified by the ◦Final Draft International Standard of the ◦ISO/ANSI standardization committee in November 1997. In all likelihood, this means I use a few features your compilers don't yet support. Don't worry. The only "new" feature I assume you have is templates, and templates are now almost universally available. I use exceptions, too, but that use is largely confined to Items 9 through 15, which are specifically devoted to exceptions. If you don't have access to a compiler offering exceptions, that's okay. It won't affect your ability to take advantage of the material in the other parts of the book. Furthermore, you should read Items 9 through 15 even if you don't have support for exceptions, because those items (as well as the associated articles) examine issues you need to understand in any case.

I recognize that just because the standardization committee blesses a feature or endorses a practice, there's no guarantee that the feature is present in current compilers or the practice is applicable to existing environments. When faced with a discrepancy between theory (what the committee says) and practice (what actually works), I discuss both, though my bias is toward things that work. Because I discuss both, this book will aid you as your compilers approach conformance with the standard. It will show you how to use existing constructs to approximate language features your compilers don't yet support, and it will guide you when you decide to transform workarounds into newly- supported features.

Notice that I refer to your *compilers* — plural. Different compilers implement varying approximations to the standard, so I encourage you to develop your code under at least two compilers. Doing so will help you avoid inadvertent dependence on one vendor's proprietary language extension or its misinterpretation of the standard. It will also help keep you away from the bleeding edge of compiler technology, e.g., from new features supported by only one vendor. Such features are often poorly implemented (buggy or slow — frequently both), and upon their introduction, the C++ community lacks experience to advise you in their proper use. Blazing trails can be exciting, but when your goal is producing reliable code, it's often best to let others test the

waters before jumping in.

There are two constructs you'll see in this book that may not be familiar to you. Both are relatively recent language extensions. Some compilers support them, but if your compilers don't, you can easily approximate them with features you do have.

The first construct is the `bool` type, which has as its values the keywords `true` and `false`. If your compilers haven't implemented `bool`, there are two ways to approximate it. One is to use a global enum:

```
enum bool { false, true };
```

This allows you to overload functions on the basis of whether they take a `bool` or an `int`, but it has the disadvantage that the built-in comparison operators (i.e., ==, <, >=, etc.) still return `int`s. As a result, code like the following will not behave the way it's supposed to:

```
void f(int);
void f(bool);

int x, y;
...
f( x < y );                              // calls f(int), but it
                                         // should call f(bool)
```

The enum approximation may thus lead to code whose behavior changes when you submit it to a compiler that truly supports `bool`.

An alternative is to use a typedef for `bool` and constant objects for `true` and `false`:

```
typedef int bool;

const bool false = 0;
const bool true = 1;
```

This is compatible with the traditional semantics of C and C++, and the behavior of programs using this approximation won't change when they're ported to `bool`-supporting compilers. The drawback is that you can't differentiate between `bool` and `int` when overloading functions. Both approximations are reasonable. Choose the one that best fits your circumstances.

The second new construct is really four constructs, the casting forms `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. If you're not familiar with these casts, you'll want to turn to Item 2 and read all about them. Not only do they do more than the C-style casts they replace, they do it better. I use these new casting forms whenever I need to perform a cast in this book.

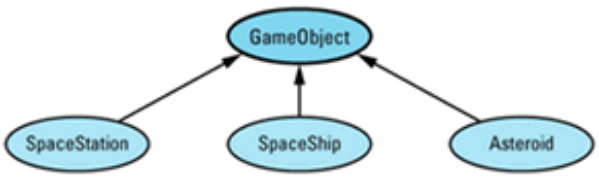
There is more to C++ than the language itself. There is also the standard library (see Item E49). Where possible, I employ the standard `string` type instead of using raw `char*` pointers, and I encourage you to do the same. `string` objects are no more difficult to manipulate than `char*`-based strings, and they relieve you of most memory-management concerns. Furthermore, `string` objects are less susceptible to memory leaks if an exception is thrown (see Items 9 and 10). A well-implemented `string` type can hold its own in an efficiency contest with its `char*` equivalent, and it may even do better. (For insight into how this could be, see Item 29.) If you don't have access to an implementation of the standard `string` type, you almost certainly have access to *some* `string`-like class. Use it. Just about anything is preferable to raw `char*`s.

I use data structures from the standard library whenever I can. Such data structures are drawn from the Standard Template Library (the "STL" — see Item 35). The STL includes bitsets, vectors, lists, queues, stacks, maps, sets, and more, and you should prefer these standardized data structures to the ad hoc equivalents you might otherwise be tempted to write. Your compilers may not have the STL bundled in, but don't let that keep you from using it. Thanks to Silicon Graphics, you can download a free copy that works with many compilers from the ◦SGI STL web site.

If you currently use a library of algorithms and data structures and are happy with it, there's no need to switch to the STL just because it's "standard." However, if you have a choice between using an STL component or writing your own code from scratch, you should lean toward using the STL. Remember code reuse? STL (and the rest of the standard library) has lots of code that is very much worth reusing.

Conventions and Terminology

Any time I mention inheritance in this book, I mean public inheritance (see Item E35). If I don't mean public inheritance, I'll say so explicitly. When drawing inheritance hierarchies, I depict base-derived relationships by drawing arrows from derived classes to base classes. For example, here is a hierarchy from Item 31:



This notation is the reverse of the convention I employed in the first (but not the second) edition of *Effective C++*. I'm now convinced that most C++ practitioners draw inheritance arrows from derived to base classes, and I am happy to follow suit. Within such diagrams, abstract classes (e.g., `GameObject`) are shaded and concrete classes (e.g., `SpaceShip`) are unshaded.

Inheritance gives rise to pointers and references with two different types, a *static type* and a *dynamic type*. The static type of a pointer or reference is its *declared* type. The dynamic type is determined by the type of object it actually *refers* to. Here are some examples based on the classes above:

```
GameObject *pgo =                       // static type of pgo is
  new SpaceShip;                        // GameObject*, dynamic
                                        // type is SpaceShip*

Asteroid *pa = new Asteroid;            // static type of pa is
                                        // Asteroid*. So is its
                                        // dynamic type

pgo = pa;                               // static type of pgo is
                                        // still (and always)
                                        // GameObject*. Its
                                        // dynamic type is now
                                        // Asteroid*

GameObject& rgo = *pa;                  // static type of rgo is
                                        // GameObject, dynamic
                                        // type is Asteroid
```

These examples also demonstrate a naming convention I like. `pgo` is a pointer-to-`GameObject`; `pa` is a pointer-to-`Asteroid`; `rgo` is a reference-to-`GameObject`. I often concoct pointer and reference names in this fashion.

Two of my favorite parameter names are `lhs` and `rhs`, abbreviations for "left-hand side" and "right-hand side," respectively. To understand the rationale behind these names, consider a class for representing rational numbers:

```
class Rational { ... };
```

If I wanted a function to compare pairs of `Rational` objects, I'd declare it like this:

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

That would let me write this kind of code:

```
Rational r1, r2;

...

if (r1 == r2) ...
```

Within the call to `operator==`, `r1` appears on the left-hand side of the "==" and is bound to `lhs`, while `r2` appears on the right-hand side of the "==" and is bound to `rhs`.

Other abbreviations I employ include *ctor* for "constructor," *dtor* for "destructor," and *RTTI* for C++'s support for runtime type identification (of which `dynamic_cast` is the most commonly used component).

When you allocate memory and fail to free it, you have a memory leak. Memory leaks arise in both C and C++, but in C++, memory leaks leak more than just memory. That's because C++ automatically calls constructors when objects are created, and constructors may themselves allocate resources. For example, consider this code:

```
class Widget { ... };                    // some class — it doesn't
                                         // matter what it is

Widget *pw = new Widget;                 // dynamically allocate a
                                         // Widget object

...                                      // assume pw is never
                                         // deleted
```

This code leaks memory, because the `Widget` pointed to by `pw` is never deleted. However, if the `Widget` constructor allocates additional resources that are to be released when the `Widget` is destroyed (such as file descriptors, semaphores, window handles, database locks, etc.), those resources are lost just as surely as the memory is. To emphasize that memory leaks in C++ often leak other resources, too, I usually speak of *resource leaks* in this book rather than memory leaks.

You won't see many inline functions in this book. That's not because I dislike inlining. Far from it, I believe that inline functions are an important feature of C++. However, the criteria for determining whether a function should be inlined can be complex, subtle, and platform-dependent (see Item E33). As a result, I avoid inlining unless there is a point about inlining I wish to make. When you see a non-inline function in *More Effective C++*, that doesn't mean I think it would be a bad idea to declare the function `inline`, it just means the decision to inline that function is independent of the material I'm examining at that point in the book.

A few C++ features have been *deprecated* by the standardization committee. Such features are slated for eventual removal from the language, because newer features have been added that do what the deprecated features do, but do it better. In this book, I identify deprecated constructs and explain what features replace them. You should try to avoid deprecated features where you can, but there's no reason to be overly concerned about their use. In the interest of preserving backward compatibility for their customers, compiler vendors are likely to support deprecated features for many years.

A *client* is somebody (a programmer) or something (a class or function, typically) that uses the code you write. For example, if you write a `Date` class (for representing birthdays, deadlines, when the Second Coming occurs, etc.), anybody using that class is your client. Furthermore, any sections of code that use the `Date` class are your clients as well. Clients are important. In fact, clients are the name of the game! If nobody uses the software you write, why write it? You will find I worry a lot about making things easier for clients, often at the expense of making things more difficult for you, because good software is "clientcentric" — it revolves around clients. If this strikes you as unreasonably philanthropic, view it instead through a lens of self-interest. Do you ever use the classes or functions you write? If so, you're your own client, so making things easier for clients in general also makes them easier for you.

When discussing class or function templates and the classes or functions generated from them, I reserve the right to be sloppy about the difference between the templates and their instantiations. For example, if `Array` is a class template taking a type parameter `T`, I may refer to a particular instantiation of the template as an `Array`, even though `Array<T>` is really the name of the class. Similarly, if `swap` is a function template taking a type parameter `T`, I may refer to an instantiation as `swap` instead of `swap<T>`. In cases where this kind of shorthand might be unclear, I include template parameters when referring to template instantiations.

Reporting Bugs, Making Suggestions, Getting Book Updates

I have tried to make this book as accurate, readable, and useful as possible, but I know there is room for improvement. If you find an error of any kind — technical, grammatical, typographical, *whatever* — please tell me about it. I will try to correct the mistake in future printings of the book, and if you are the first person to report it, I will gladly add your name to the book's acknowledgments. If you have other suggestions for improvement, I welcome those, too.

I continue to collect guidelines for effective programming in C++. If you have ideas for new guidelines, I'd be delighted if you'd share them with me. Send your guidelines, your comments, your criticisms, and your bug reports to:

> Scott Meyers
> c/o Editor-in-Chief, Corporate and Professional Publishing
> Addison-Wesley Publishing Company
> 1 Jacob Way
> Reading, MA 01867
> U. S. A.

Alternatively, you may send electronic mail to `mec++@awl.com`.

I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. This list, along with other book-related information, is available from the ◦Web site for this book. It is also available via anonymous FTP from ◦`ftp.awl.com` in the directory `cp/mec++`. If you would like a copy of the list of changes to this book, but you lack access to the Internet, please send a request to one of the addresses above, and I will see that the list is sent to

you.

Enough preliminaries. On with the show!

Back to Acknowledgments
Continue to Basics

Back to Introduction
Continue to Item 1: Distinguish between pointers and references

Basics

Ah, the basics. Pointers, references, casts, arrays, constructors — you can't get much more basic than that. All but the simplest C++ programs use most of these features, and many programs use them all.

In spite of our familiarity with these parts of the language, sometimes they can still surprise us. This is especially true for programmers making the transition from C to C++, because the concepts behind references, dynamic casts, default constructors, and other non-C features are usually a little murky.

This chapter describes the differences between pointers and references and offers guidance on when to use each. It introduces the new C++ syntax for casts and explains why the new casts are superior to the C-style casts they replace. It examines the C notion of arrays and the C++ notion of polymorphism, and it describes why mixing the two is an idea whose time will never come. Finally, it considers the pros and cons of default constructors and suggests ways to work around language restrictions that encourage you to have one when none makes sense.

By heeding the advice in the items that follow, you'll make progress toward a worthy goal: producing software that expresses your design intentions clearly and correctly.

Back to Introduction
Continue to Item 1: Distinguish between pointers and references

Back to Basics
Continue to Item 2: Prefer C++-style casts

Item 1: Distinguish between pointers and references.

Pointers and references *look* different enough (pointers use the "*" and "->" operators, references use "."), but they seem to do similar things. Both pointers and references let you refer to other objects indirectly. How, then, do you decide when to use one and not the other?

First, recognize that there is no such thing as a null reference. A reference must *always* refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must *always* refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference.

"But wait," you wonder, "what about underhandedness like this?"

```
char *pc = 0;          // set pointer to null

char& rc = *pc;        // make reference refer to
                       // dereferenced null pointer
```

Well, this is evil, pure and simple. The results are undefined (compilers can generate output to do anything they like), and people who write this kind of code should be shunned until they agree to cease and desist. If you have to worry about things like this in your software, you're probably best off avoiding references entirely. Either that or finding a better class of programmers to work with. We'll henceforth ignore the possibility that a reference can be "null."

Because a reference must refer to an object, C++ requires that references be initialized:

```
string& rs;             // error! References must
                        // be initialized

string s("xyzzy");

string& rs = s;         // okay, rs refers to s
```

Pointers are subject to no such restriction:

```
string *ps;             // uninitialized pointer:
                        // valid but risky
```

The fact that there is no such thing as a null reference implies that it can be more efficient to use references than to use pointers. That's because there's no need to test the validity of a reference before using it:

```
void printDouble(const double& rd)
{
    cout << rd;         // no need to test rd; it
}                       // must refer to a double
```

Pointers, on the other hand, should generally be tested against null:

```
void printDouble(const double *pd)
{
  if (pd) {             // check for null pointer
    cout << *pd;
  }
}
```

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. A reference, however, *always* refers to the object with which it is initialized:

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;         // rs refers to s1

string *ps = &s1;        // ps points to s1

rs = s2;                 // rs still refers to s1,
                         // but s1's value is now
                         // "Clancy"

ps = &s2;                // ps now points to s2;
                         // s1 is unchanged
```

In general, you should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points). You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else.

There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is `operator[]`. This operator typically needs to return something that can be used as the target of an assignment:

```
vector<int> v(10);       // create an int vector of size 10;
                         // vector is a template in the
                         // standard C++ library (see Item 35)

v[5] = 10;               // the target of this assignment is
                         // the return value of operator[]
```

If `operator[]` returned a pointer, this last statement would have to be written this way:

```
*v[5] = 10;
```

But this makes it look like `v` is a vector of pointers, which it's not. For this reason, you'll almost always want `operator[]` to return a reference. (For an interesting exception to this rule, see Item 30.)

References, then, are the feature of choice when you *know* you have something to refer to, when you'll *never* want to refer to anything else, and when implementing operators whose syntactic requirements make the use of pointers undesirable. In all other cases, stick with pointers.

Back to Basics
Continue to Item 2: Prefer C++-style casts

Back to Item 1: Distinguish between pointers and references
Continue to Item 3: Never treat arrays polymorphically

Item 2: Prefer C++-style casts.

Consider the lowly cast. Nearly as much a programming pariah as the `goto`, it nonetheless endures, because when worse comes to worst and push comes to shove, casts can be necessary. Casts are especially necessary when worse comes to worst and push comes to shove.

Still, C-style casts are not all they might be. For one thing, they're rather crude beasts, letting you cast pretty much any type to pretty much any other type. It would be nice to be able to specify more precisely the purpose of each cast. There is a great difference, for example, between a cast that changes a pointer-to-`const`-object into a pointer-to-non-`const`-object (i.e., a cast that changes only the `const`ness of an object) and a cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (i.e., a cast that completely changes an object's type). Traditional C-style casts make no such distinctions. (This is hardly a surprise. C-style casts were designed for C, not C++.)

A second problem with casts is that they are hard to find. Syntactically, casts consist of little more than a pair of parentheses and an identifier, and parentheses and identifiers are used everywhere in C++. This makes it tough to answer even the most basic cast-related questions, questions like, "Are any casts used in this program?" That's because human readers are likely to overlook casts, and tools like `grep` cannot distinguish them from non-cast constructs that are syntactically similar.

C++ addresses the shortcomings of C-style casts by introducing four new cast operators, `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. For most purposes, all you need to know about these operators is that what you are accustomed to writing like this,

```
(type) expression
```

you should now generally write like this:

```
static_cast<type>(expression)
```

For example, suppose you'd like to cast an `int` to a `double` to force an expression involving `int`s to yield a floating point value. Using C-style casts, you could do it like this:

```
int firstNumber, secondNumber;

...

double result = ((double)firstNumber)/secondNumber;
```

With the new casts, you'd write it this way:

```
double result = static_cast<double>(firstNumber)/secondNumber;
```

Now *there's* a cast that's easy to see, both for humans and for programs.

`static_cast` has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, you can't cast a `struct` into an `int` or a `double` into a pointer using `static_cast` any more than you can with a C-style cast. Furthermore, `static_cast` can't remove `const`ness from an expression, because another new cast, `const_cast`, is designed specifically to do that.

The other new C++ casts are used for more restricted purposes. `const_cast` is used to cast away the `const`ness or `volatile`ness of an expression. By using a `const_cast`, you emphasize (to both humans and compilers) that the only thing you want to change through the cast is the `const`ness or `volatile`ness of something. This meaning is enforced by compilers. If you try to employ `const_cast` for anything other than modifying the `const`ness or `volatile`ness of an expression, your cast will be rejected. Here are some examples:

```
class Widget { ... };
class SpecialWidget: public Widget { ... };


void update(SpecialWidget *psw);


SpecialWidget sw;                       // sw is a non-const object,
const SpecialWidget& csw = sw;          // but csw is a reference to
                                        // it as a const object

update(&csw);           // error! can't pass a const
                        // SpecialWidget* to a function
                        // taking a SpecialWidget*

update(const_cast<SpecialWidget*>(&csw));
                        // fine, the constness of &csw is
                        // explicitly cast away (and
                        // csw — and sw — may now be
                        // changed inside update)

update((SpecialWidget*)&csw);
                        // same as above, but using a
                        // harder-to-recognize C-style cast

Widget *pw = new SpecialWidget;
```

```
update(pw);             // error! pw's type is Widget*, but
                        // update takes a SpecialWidget*

update(const_cast<SpecialWidget*>(pw));
                        // error! const_cast can be used only
                        // to affect constness or volatileness,
                        // never to cast down the inheritance
                        // hierarch
```

By far the most common use of `const_cast` is to cast away the `const`ness of an object.

The second specialized type of cast, `dynamic_cast`, is used to perform *safe casts* down or across an inheritance hierarchy. That is, you use `dynamic_cast` to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts succeeded.[1] Failed casts are indicated by a null pointer (when casting pointers) or an exception (when casting references):

```
Widget *pw;


...


update(dynamic_cast<SpecialWidget*>(pw));
                        // fine, passes to update a pointer
                        // to the SpecialWidget pw points to
                        // if pw really points to one,
                        // otherwise passes the null pointer


void updateViaRef(SpecialWidget& rsw);


updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                        // fine, passes to updateViaRef the
                        // SpecialWidget pw points to if pw
                        // really points to one, otherwise
                        // throws an exception
```

`dynamic_casts` are restricted to helping you navigate inheritance hierarchies. They cannot be applied to types lacking virtual functions (see also Item 24), nor can they cast away `const`ness:

```
int firstNumber, secondNumber;
...
double result = dynamic_cast<double>(firstNumber)/secondNumber;
```

```
                          // error! no inheritance is involved


const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                          // error! dynamic_cast can't cast
                          // away constness
```

If you want to perform a cast on a type where inheritance is not involved, you probably want a `static_cast`. To cast `const`ness away, you always want a `const_cast`.

The last of the four new casting forms is `reinterpret_cast`. This operator is used to perform type conversions whose result is nearly always implementation-defined. As a result, `reinterpret_casts` are rarely portable.

The most common use of `reinterpret_cast` is to cast between function pointer types. For example, suppose you have an array of pointers to functions of a particular type:

```
typedef void (*FuncPtr)();         // a FuncPtr is a pointer
                                   // to a function taking no
                                   // args and returning void


FuncPtr funcPtrArray[10];          // funcPtrArray is an array
                                   // of 10 FuncPtrs
```

Let us suppose you wish (for some unfathomable reason) to place a pointer to the following function into `funcPtrArray`:

```
int doSomething();
```

You can't do what you want without a cast, because `doSomething` has the wrong type for `funcPtrArray`. The functions in `funcPtrArray` return `void`, but `doSomething` returns an `int`:

```
funcPtrArray[0] = &doSomething;     // error! type mismatch
```

A `reinterpret_cast` lets you force compilers to see things your way:

```
funcPtrArray[0] =                   // this compiles
```

```
reinterpret_cast<FuncPtr>(&doSomething);
```

Casting function pointers is not portable (C++ offers no guarantee that all function pointers are represented the same way), and in some cases such casts yield incorrect results (see Item 31), so you should avoid casting function pointers unless your back's to the wall and a knife's at your throat. A sharp knife. A *very* sharp knife.

If your compilers lack support for the new casting forms, you can use traditional casts in place of `static_cast`, `const_cast`, and `reinterpret_cast`. Furthermore, you can use macros to approximate the new syntax:

```
#define static_cast(TYPE,EXPR)       ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)        ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)  ((TYPE)(EXPR))
```

You'd use the approximations like this:

```
double result = static_cast(double, firstNumber)/secondNumber;


update(const_cast(SpecialWidget*, &sw));


funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

These approximations won't be as safe as the real things, of course, but they will simplify the process of upgrading your code when your compilers support the new casts.

There is no easy way to emulate the behavior of a `dynamic_cast`, but many libraries provide functions to perform safe inheritance-based casts for you. If you lack such functions and you *must* perform this type of cast, you can fall back on C-style casts for those, too, but then you forego the ability to tell if the casts fail. Needless to say, you can define a macro to look like `dynamic_cast`, just as you can for the other casts:

```
#define dynamic_cast(TYPE,EXPR)     (TYPE)(EXPR)
```

Remember that this approximation is not performing a true `dynamic_cast`; there is no way to tell if the cast fails.

I know, I know, the new casts are ugly and hard to type. If you find them too unpleasant to look at, take solace in the knowledge that C-style casts continue to be valid. However, what the new casts lack in beauty they make up for in precision of meaning and easy recognizability. Programs that use the new casts are easier to parse (both for humans and for tools), and they allow compilers to diagnose casting errors that would otherwise go undetected. These are powerful arguments for abandoning C-style casts, and there may also be a third: perhaps making casts ugly and hard to type is a *good* thing.

Back to Item 1: Distinguish between pointers and references
Continue to Item 3: Never treat arrays polymorphically

---

[1] A second, unrelated use of `dynamic_cast` is to find the beginning of the memory occupied by an object. We explore that capability in Item 27.
Return

Back to Item 2: Prefer C++-style casts
Continue to Item 4: Avoid gratuitous default constructors

Item 3: Never treat arrays polymorphically.

One of the most important features of inheritance is that you can manipulate derived class objects through pointers and references to base class objects. Such pointers and references are said to behave *polymorphically* — as if they had multiple types. C++ also allows you to manipulate *arrays* of derived class objects through base class pointers and references. This is no feature at all, because it almost never works the way you want it to.

For example, suppose you have a class `BST` (for binary search tree objects) and a second class, `BalancedBST`, that inherits from `BST`:

```
class BST { ... };

class BalancedBST: public BST { ... };
```

In a real program such classes would be templates, but that's unimportant here, and adding all the template syntax just makes things harder to read. For this discussion, we'll assume `BST` and `BalancedBST` objects contain only `int`s.

Consider a function to print out the contents of each `BST` in an array of `BST`s:

```
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];              // this assumes an
  }                             // operator<< is defined
}                               // for BST objects
```

This will work fine when you pass it an array of `BST` objects:

```
BST BSTArray[10];

...

printBSTArray(cout, BSTArray, 10);          // works fine
```

Consider, however, what happens when you pass `printBSTArray` an array of `BalancedBST` objects:

```
BalancedBST bBSTArray[10];

...

printBSTArray(cout, bBSTArray, 10);         // works fine?
```

Your compilers will accept this function call without complaint, but look again at the loop for which they must generate code:

```
for (int i = 0; i < numElements; ++i) {
  s << array[i];
}
```

Now, `array[i]` is really just shorthand for an expression involving pointer arithmetic: it stands for `*(array+i)`. We know that `array` is a pointer to the beginning of the array, but how far away from the memory location pointed to by `array` is the memory location pointed to by `array+i`? The distance between them is `i*sizeof(`*an object in the array*`)`, because there are `i` objects between `array[0]` and `array[i]`. In order for compilers to emit code that walks through the array correctly, they must be able to determine the size of the objects in the array. This is easy for them to do. The parameter `array` is declared to be of type array-of-`BST`, so each element of the array must be a `BST`, and the distance between `array` and `array+i` must be `i*sizeof(BST)`.

At least that's how your compilers look at it. But if you've passed an array of `BalancedBST` objects to `printBSTArray`, your compilers are probably wrong. In that case, they'd assume each object in the array is the size of a `BST`, but each object would actually be the size of a `BalancedBST`. Derived classes usually have more data members than their base classes, so derived class objects are usually larger than base class objects. We thus expect a `BalancedBST` object to be larger than a `BST` object. If it is, the pointer arithmetic generated for `printBSTArray` will be wrong for arrays of `BalancedBST` objects, and there's no telling what will happen when `printBSTArray` is invoked on a `BalancedBST` array. Whatever does happen, it's a good bet it won't be pleasant.

The problem pops up in a different guise if you try to delete an array of derived class objects through a base class pointer. Here's one way you might innocently attempt to do it:

```
// delete an array, but first log a message about its
// deletion
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << '\n';

  delete [] array;
}

BalancedBST *balTreeArray =                 // create a BalancedBST
  new BalancedBST[50];                      // array
```

```
...

deleteArray(cout, balTreeArray);            // log its deletion
```

You can't see it, but there's pointer arithmetic going on here, too. When an array is deleted, a destructor for each element of the array must be called (see Item 8). When compilers see the statement

```
delete [] array;
```

they must generate code that does something like this:

```
// destruct the objects in *array in the inverse order
// in which they were constructed
for (int i = the number of elements in the array - 1;
     i >= 0;
     --i)
{
  array[i].BST::~BST();                    // call array[i]'s
}                                          // destructor
```

Just as this kind of loop failed to work when you wrote it, it will fail to work when your compilers write it, too. The language specification says the result of deleting an array of derived class objects through a base class pointer is undefined, but we know what that really means: executing the code is almost certain to lead to grief. Polymorphism and pointer arithmetic simply don't mix. Array operations almost always involve pointer arithmetic, so arrays and polymorphism don't mix.

Note that you're unlikely to make the mistake of treating an array polymorphically if you avoid having a concrete class (like `BalancedBST`) inherit from another concrete class (such as `BST`). As Item 33 explains, designing your software so that concrete classes never inherit from one another has many benefits. I encourage you to turn to Item 33 and read all about them.

Back to Item 2: Prefer C++-style casts
Continue to Item 4: Avoid gratuitous default constructors

Back to Item 3: Never treat arrays polymorphically
Continue to Operators

Item 4: Avoid gratuitous default constructors.

A default constructor (i.e., a constructor that can be called with no arguments) is the C++ way of saying you can get something for nothing. Constructors initialize objects, so default constructors initialize objects without any information from the place where the object is being created. Sometimes this makes perfect sense. Objects that act like numbers, for example, may reasonably be initialized to zero or to undefined values. Objects that act like pointers (Item 28) may reasonably be initialized to null or to undefined values. Data structures like linked lists, hash tables, maps, and the like may reasonably be initialized to empty containers.

Not all objects fall into this category. For many objects, there is no reasonable way to perform a complete initialization in the absence of outside information. For example, an object representing an entry in an address book makes no sense unless the name of the thing being entered is provided. In some companies, all equipment must be tagged with a corporate ID number, and creating an object to model a piece of equipment in such companies is nonsensical unless the appropriate ID number is provided.

In a perfect world, classes in which objects could reasonably be created from nothing would contain default constructors and classes in which information was required for object construction would not. Alas, ours is not the best of all possible worlds, so we must take additional concerns into account. In particular, if a class lacks a default constructor, there are restrictions on how you can use that class.

Consider a class for company equipment in which the corporate ID number of the equipment is a mandatory constructor argument:

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
  ...
};
```

Because `EquipmentPiece` lacks a default constructor, its use may be problematic in three contexts. The first is the creation of arrays. There is, in general, no way to specify constructor arguments for objects in arrays, so it is not usually possible to create arrays of `EquipmentPiece` objects:

```
EquipmentPiece bestPieces[10];          // error! No way to call
                                        // EquipmentPiece ctors

EquipmentPiece *bestPieces =
```

```
  new EquipmentPiece[10];                // error! same problem
```

There are three ways to get around this restriction. A solution for non-heap arrays is to provide the necessary arguments at the point where the array is defined:

```
int ID1, ID2, ID3, ..., ID10;            // variables to hold
                                         // equipment ID numbers
...

EquipmentPiece bestPieces[] = {          // fine, ctor arguments
  EquipmentPiece(ID1),                   // are provided
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

Unfortunately, there is no way to extend this strategy to heap arrays.

A more general approach is to use an array of *pointers* instead of an array of objects:

```
typedef EquipmentPiece* PEP;             // a PEP is a pointer to
                                         // an EquipmentPiece

PEP bestPieces[10];                      // fine, no ctors called


PEP *bestPieces = new PEP[10];           // also fine
```

Each pointer in the array can then be made to point to a different `EquipmentPiece` object:

```
for (int i = 0; i < 10; ++i)
  bestPieces[i] = new EquipmentPiece( ID Number );
```

There are two disadvantages to this approach. First, you have to remember to delete all the objects pointed to by the array. If you forget, you have a resource leak. Second, the total amount of memory you need increases, because you need the space for the pointers as well as the space for the `EquipmentPiece` objects.

You can avoid the space penalty if you allocate the raw memory for the array, then use "placement `new`" (see Item 8) to construct the `EquipmentPiece` objects in the memory:

```
// allocate enough raw memory for an array of 10
// EquipmentPiece objects; see Item 8 for details on
// the operator new[] function
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));


// make bestPieces point to it so it can be treated as an
// EquipmentPiece array
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);


// construct the EquipmentPiece objects in the memory
// using "placement new" (see Item 8)
for (int i = 0; i < 10; ++i)
  new (&bestPieces[i]) EquipmentPiece( ID Number );
```

Notice that you still have to provide a constructor argument for each `EquipmentPiece` object. This technique (as well as the array-of-pointers idea) allows you to create arrays of objects when a class lacks a default constructor; it doesn't show you how to bypass required constructor arguments. There is no way to do that. If there were, it would defeat the purpose of constructors, which is to *guarantee* that objects are initialized.

The downside to using placement `new`, aside from the fact that most programmers are unfamiliar with it (which will make maintenance more difficult), is that you must manually call destructors on the objects in the array when you want them to go out of existence, then you must manually deallocate the raw memory by calling `operator delete[]` (again, see Item 8):

```
// destruct the objects in bestPieces in the inverse
// order in which they were constructed
for (int i = 9; i >= 0; --i)
  bestPieces[i].~EquipmentPiece();

// deallocate the raw memory
operator delete[](rawMemory);
```

If you forget this requirement and use the normal array-deletion syntax, your program will behave unpredictably. That's because the result of deleting a pointer that didn't come from the `new` operator is undefined:

```
delete [] bestPieces;                    // undefined! bestPieces
                                         // didn't come from the new
                                         // operator
```

For more information on the `new` operator, placement `new` and how they interact with constructors

and destructors, see Item 8.

The second problem with classes lacking default constructors is that they are ineligible for use with many template-based container classes. That's because it's a common requirement for such templates that the type used to instantiate the template provide a default constructor. This requirement almost always grows out of the fact that inside the template, an array of the template parameter type is being created. For example, a template for an `Array` class might look something like this:

```
template<class T>
class Array {
public:
  Array(int size);
  ...

private:
  T *data;
};

template<class T>
Array<T>::Array(int size)
{
  data = new T[size];                    // calls T::T() for each
  ...                                    // element of the array
}
```

In most cases, careful template design can eliminate the need for a default constructor. For example, the standard `vector` template (which generates classes that act like extensible arrays) has no requirement that its type parameter have a default constructor. Unfortunately, many templates are designed in a manner that is anything but careful. That being the case, classes without default constructors will be incompatible with many templates. As C++ programmers learn more about template design, this problem should recede in significance. How long it will take for that to happen, however, is anyone's guess.

The final consideration in the to-provide-a-default-constructor-or-not-to-provide-a-default-constructor dilemma has to do with virtual base classes (see Item E43). Virtual base classes lacking default constructors are a pain to work with. That's because the arguments for virtual base class constructors must be provided by the most derived class of the object being constructed. As a result, a virtual base class lacking a default constructor requires that *all* classes derived from that class — no matter how far removed — must know about, understand the meaning of, and provide for the virtual base class's constructors' arguments. Authors of derived classes neither expect nor appreciate this requirement.

Because of the restrictions imposed on classes lacking default constructors, some people believe *all* classes should have them, even if a default constructor doesn't have enough information to fully initialize objects of that class. For example, adherents to this philosophy might modify `EquipmentPiece` as follows:

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber = UNSPECIFIED);
  ...


private:
  static const int UNSPECIFIED;          // magic ID number value
                                         // meaning no ID was
};                                       // specified
```

This allows `EquipmentPiece` objects to be created like this:

```
EquipmentPiece e;                         // now okay
```

Such a transformation almost always complicates the other member functions of the class, because there is no longer any guarantee that the fields of an `EquipmentPiece` object have been meaningfully initialized. Assuming it makes no sense to have an `EquipmentPiece` without an ID field, most member functions must check to see if the ID is present. If it's not, they'll have to figure out how to stumble on anyway. Often it's not clear how to do that, and many implementations choose a solution that offers nothing but expediency: they throw an exception or they call a function that terminates the program. When that happens, it's difficult to argue that the overall quality of the software has been improved by including a default constructor in a class where none was warranted.

Inclusion of meaningless default constructors affects the efficiency of classes, too. If member functions have to test to see if fields have truly been initialized, clients of those functions have to pay for the time those tests take. Furthermore, they have to pay for the code that goes into those tests, because that makes executables and libraries bigger. They also have to pay for the code that handles the cases where the tests fail. All those costs are avoided if a class's constructors ensure that all fields of an object are correctly initialized. Often default constructors can't offer that kind of assurance, so it's best to avoid them in classes where they make no sense. That places some limits on how such classes can be used, yes, but it also guarantees that when you do use such classes, you can expect that the objects they generate are fully initialized and are efficiently implemented.

Back to Item 3: Never treat arrays polymorphically
Continue to Operators

Back to Item 4: Avoid gratuitous default constructors
Continue to Item 5: Be wary of user-defined conversion functions

Operators

Overloadable operators — you gotta love 'em! They allow you to give your types the same syntax as C++'s built-in types, yet they let you put a measure of power into the functions *behind* the operators that's unheard of for the built-ins. Of course, the fact that you can make symbols like "+" and "==" do anything you want also means you can use overloaded operators to produce programs best described as impenetrable. Adept C++ programmers know how to harness the power of operator overloading without descending into the incomprehensible.

Regrettably, it is easy to make the descent. Single-argument constructors and implicit type conversion operators are particularly troublesome, because they can be invoked without there being any source code showing the calls. This can lead to program behavior that is difficult to understand. A different problem arises when you overload operators like `&&` and `||`, because the shift from built-in operator to user-defined function yields a subtle change in semantics that's easy to overlook. Finally, many operators are related to one another in standard ways, but the ability to overload operators makes it possible to violate the accepted relationships.

In the items that follow, I focus on explaining when and how overloaded operators are called, how they behave, how they should relate to one another, and how you can seize control of these aspects of overloaded operators. With the information in this chapter under your belt, you'll be overloading (or *not* overloading) operators like a pro.

Back to Item 4: Avoid gratuitous default constructors
Continue to Item 5:Be wary of user-defined conversion functions

Back to Operators
Continue to Item 6: Distinguish between prefix and postfix forms of increment and decrement operators

Item 5: Be wary of user-defined conversion functions.

C++ allows compilers to perform implicit conversions between types. In honor of its C heritage, for example, the language allows silent conversions from `char` to `int` and from `short` to `double`. This is why you can pass a `short` to a function that expects a `double` and still have the call succeed. The more frightening conversions in C — those that may lose information — are also present in C++, including conversion of `int` to `short` and `double` to (of all things) `char`.

You can't do anything about such conversions, because they're hard-coded into the language. When you add your own types, however, you have more control, because you can choose whether to provide the functions compilers are allowed to use for implicit type conversions.

Two kinds of functions allow compilers to perform such conversions: *single-argument constructors* and *implicit type conversion operators*. A single-argument constructor is a constructor that may be called with only one argument. Such a constructor may declare a single parameter or it may declare multiple parameters, with each parameter after the first having a default value. Here are two examples:

```
class Name {                                  // for names of things
public:
  Name(const string& s);                      // converts string to
                                              // Name
  ...

};

class Rational {                              // for rational numbers
public:
  Rational(int numerator = 0,                 // converts int to
           int denominator = 1);              // Rational
  ...

};
```

An implicit type conversion operator is simply a member function with a strange-looking name: the word `operator` followed by a type specification. You aren't allowed to specify a type for the function's return value, because the type of the return value is basically just the name of the function. For example, to allow `Rational` objects to be implicitly converted to `doubles` (which might be useful for mixed-mode arithmetic involving `Rational` objects), you might define class `Rational` like this:

```
class Rational {
public:
```

```
  ...
  operator double() const;                    // converts Rational to
};                                            // double
```

This function would be automatically invoked in contexts like this:

```
Rational r(1, 2);                             // r has the value 1/2

double d = 0.5 * r;                           // converts r to a double,
                                              // then does multiplication
```

Perhaps all this is review. That's fine, because what I really want to explain is why you usually don't want to provide type conversion functions of *any* ilk.

The fundamental problem is that such functions often end up being called when you neither want nor expect them to be. The result can be incorrect and unintuitive program behavior that is maddeningly difficult to diagnose.

Let us deal first with implicit type conversion operators, as they are the easiest case to handle. Suppose you have a class for rational numbers similar to the one above, and you'd like to print `Rational` objects as if they were a built-in type. That is, you'd like to be able to do this:

```
Rational r(1, 2);

cout << r;                                     // should print "1/2"
```

Further suppose you forgot to write an `operator<<` for `Rational` objects. You would probably expect that the attempt to print `r` would fail, because there is no appropriate `operator<<` to call. You would be mistaken. Your compilers, faced with a call to a function called `operator<<` that takes a `Rational`, would find that no such function existed, but they would then try to find an acceptable sequence of implicit type conversions they could apply to make the call succeed. The rules defining which sequences of conversions are acceptable are complicated, but in this case your compilers would discover they could make the call succeed by implicitly converting `r` to a `double` by calling `Rational::operator double`. The result of the code above would be to print `r` as a floating point number, not as a rational number. This is hardly a disaster, but it demonstrates the disadvantage of implicit type conversion operators: their presence can lead to the *wrong function* being called (i.e., one other than the one intended).

The solution is to replace the operators with equivalent functions that don't have the syntactically magic names. For example, to allow conversion of a `Rational` object to a `double`, replace `operator double` with a function called something like `asDouble`:

```
class Rational {
public:
```

```
  ...
  double asDouble() const;                    // converts Rational
};                                            // to double
```

Such a member function must be called explicitly:

```
Rational r(1, 2);

cout << r;                                    // error! No operator<<
                                              // for Rationals

cout << r.asDouble();                         // fine, prints r as a
                                              // double
```

In most cases, the inconvenience of having to call conversion functions explicitly is more than compensated for by the fact that unintended functions can no longer be silently invoked. In general, the more experience C++ programmers have, the more likely they are to eschew type conversion operators. The members of the committee working on the standard C++ library (see Item E49 and Item 35), for example, are among the most experienced in the business, and perhaps that's why the `string` type they added to the library contains no implicit conversion from a `string` object to a C-style `char*`. Instead, there's an explicit member function, `c_str`, that performs that conversion. Coincidence? I think not.

Implicit conversions via single-argument constructors are more difficult to eliminate. Furthermore, the problems these functions cause are in many cases worse than those arising from implicit type conversion operators.

As an example, consider a class template for array objects. These arrays allow clients to specify upper and lower index bounds:

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  Array(int size);

  T& operator[](int index);

  ...

};
```

The first constructor in the class allows clients to specify a range of array indices, for example, from 10 to 20. As a two-argument constructor, this function is ineligible for use as a type-conversion function. The second constructor, which allows clients to define `Array` objects by specifying only the number of elements in the array (in a manner similar to that used with built-in

arrays), is different. It *can* be used as a type conversion function, and that can lead to endless anguish.

For example, consider a template specialization for comparing `Array<int>` objects and some code that uses such objects:

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
  if (a == b[i]) {        // oops! "a" should be "a[i]"
    do something for when
    a[i] and b[i] are equal;
  }
  else {
    do something for when they're not;
  }
```

We intended to compare each element of `a` to the corresponding element in `b`, but we accidentally omitted the subscripting syntax when we typed `a`. Certainly we expect this to elicit all manner of unpleasant commentary from our compilers, but they will complain not at all. That's because they see a call to `operator==` with arguments of type `Array<int>` (for `a`) and `int` (for `b[i]`), and though there is no `operator==` function taking those types, our compilers notice they can convert the `int` into an `Array<int>` object by calling the `Array<int>` constructor that takes a single `int` as an argument. This they proceed to do, thus generating code for a program we never meant to write, one that looks like this:

```
for (int i = 0; i < 10; ++i)
  if (a == static_cast< Array<int> >(b[i]))   ...
```

Each iteration through the loop thus compares the contents of `a` with the contents of a temporary array of size `b[i]` (whose contents are presumably undefined). Not only is this unlikely to behave in a satisfactory manner, it is also tremendously inefficient, because each time through the loop we both create and destroy a temporary `Array<int>` object (see Item 19).

The drawbacks to implicit type conversion operators can be avoided by simply failing to declare the operators, but single-argument constructors cannot be so easily waved away. After all, you may really *want* to offer single-argument constructors to your clients. At the same time, you may wish to prevent compilers from calling such constructors indiscriminately. Fortunately, there is a way to

have it all. In fact, there are two ways: the easy way and the way you'll have to use if your compilers don't yet support the easy way.

The easy way is to avail yourself of one of the newest C++ features, the `explicit` keyword. This feature was introduced specifically to address the problem of implicit type conversion, and its use is about as straightforward as can be. Constructors can be declared `explicit`, and if they are, compilers are prohibited from invoking them for purposes of implicit type conversion. Explicit conversions are still legal, however:

```
template<class T>
class Array {
public:
  ...
  explicit Array(int size);                 // note use of "explicit"
  ...
};

Array<int> a(10);                           // okay, explicit ctors can
                                            // be used as usual for
                                            // object construction

Array<int> b(10);                           // also okay

if (a == b[i]) ...                          // error! no way to
                                            // implicitly convert
                                            // int to Array<int>

if (a == Array<int>(b[i])) ...              // okay, the conversion
                                            // from int to Array<int> is
                                            // explicit (but the logic of
                                            // the code is suspect)

if (a == static_cast< Array<int> >(b[i])) ...
                                            // equally okay, equally
                                            // suspect

if (a == (Array<int>)b[i]) ...              // C-style casts are also
                                            // okay, but the logic of
                                            // the code is still suspect
```

In the example using `static_cast` (see Item 2), the space separating the two ">" characters is no accident. If the statement were written like this,

```
if (a == static_cast<Array<int>>(b[i])) ...
```

it would have a different meaning. That's because C++ compilers parse ">>" as a single token. Without a space between the ">" characters, the statement would generate a syntax error.

If your compilers don't yet support `explicit`, you'll have to fall back on home-grown methods for preventing the use of single-argument constructors as implicit type conversion functions. Such methods are obvious only *after* you've seen them.

I mentioned earlier that there are complicated rules governing which sequences of implicit type conversions are legitimate and which are not. One of those rules is that no sequence of conversions is allowed to contain more than one user-defined conversion (i.e., a call to a single-argument constructor or an implicit type conversion operator). By constructing your classes properly, you can take advantage of this rule so that the object constructions you want to allow are legal, but the implicit conversions you don't want to allow are illegal.

Consider the `Array` template again. You need a way to allow an integer specifying the size of the array to be used as a constructor argument, but you must at the same time prevent the implicit conversion of an integer into a temporary `Array` object. You accomplish this by first creating a new class, `ArraySize`. Objects of this type have only one purpose: they represent the size of an array that's about to be created. You then modify `Array`'s single-argument constructor to take an `ArraySize` object instead of an `int`. The code looks like this:

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

  class ArraySize {                              // this class is new
  public:
    ArraySize(int numElements): theSize(numElements) {}
    int size() const { return theSize; }

  private:
    int theSize;
  };

  Array(int lowBound, int highBound);
  Array(ArraySize size);                     // note new declaration

  ...

};
```

Here you've nested `ArraySize` inside `Array` to emphasize the fact that it's always used in conjunction with that class. You've also made `ArraySize` public in `Array` so that anybody can use it. Good.

Consider what happens when an `Array` object is defined via the class's single-argument constructor:

```
Array<int> a(10);
```

Your compilers are asked to call a constructor in the `Array<int>` class that takes an `int`, but there is no such constructor. Compilers realize they can convert the `int` argument into a temporary `ArraySize` object, and that `ArraySize` object is just what the `Array<int>` constructor needs, so compilers perform the conversion with their usual gusto. This allows the function call (and the attendant object construction) to succeed.

The fact that you can still construct `Array` objects with an `int` argument is reassuring, but it does you little good unless the type conversions you want to avoid are prevented. They are. Consider this code again:

```
bool operator==(const Array<int>& lhs,
                const Array<int>& rhs);


Array<int> a(10);
Array<int> b(10);


...


for (int i = 0; i < 10; ++i)
  if (a == b[i]) ...                          // oops! "a" should be "a[i]";
                                              // this is now an error
```

Compilers need an object of type `Array<int>` on the right-hand side of the "==" in order to call `operator==` for `Array<int>` objects, but there is no single-argument constructor taking an `int` argument. Furthermore, compilers cannot consider converting the `int` into a temporary `ArraySize` object and then creating the necessary `Array<int>` object from this temporary, because that would call for two user-defined conversions, one from `int` to `ArraySize` and one from `ArraySize` to `Array<int>`. Such a conversion sequence is *verboten*, so compilers must issue an error for the code attempting to perform the comparison.

The use of the `ArraySize` class in this example might look like a special-purpose hack, but it's actually a specific instance of a more general technique. Classes like `ArraySize` are often called *proxy classes*, because each object of such a class stands for (is a proxy for) some other object. An `ArraySize` object is really just a stand-in for the integer used to specify the size of the `Array` being created. Proxy objects can give you control over aspects of your software's behavior — in this case implicit type conversions — that is otherwise beyond your grasp, so it's well worth your while to learn how to use them. How, you might wonder, can you acquire such learning? One way is to turn to Item 30; it's devoted to proxy classes.

Before you turn to proxy classes, however, reflect a bit on the lessons of this Item. Granting compilers license to perform implicit type conversions usually leads to more harm than good, so don't provide conversion functions unless you're *sure* you want them.

Back to Operators
Continue to Item 6: Distinguish between prefix and postfix forms of increment and decrement operators

Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.

Long, long ago (the late '80s) in a language far, far away (C++ at that time), there was no way to distinguish between prefix and postfix invocations of the ++ and -- operators. Programmers being programmers, they kvetched about this omission, and C++ was extended to allow overloading both forms of increment and decrement operators.

There was a syntactic problem, however, and that was that overloaded functions are differentiated on the basis of the parameter types they take, but neither prefix nor postfix increment or decrement takes an argument. To surmount this linguistic pothole, it was decreed that postfix forms take an `int` argument, and compilers silently pass 0 as that `int` when those functions are called:

```
class UPInt {                           // "unlimited precision int"
public:
  UPInt& operator++();                  // prefix ++
  const UPInt operator++(int);          // postfix ++

  UPInt& operator--();                  // prefix --
  const UPInt operator--(int);          // postfix --

  UPInt& operator+=(int);               // a += operator for UPInts
                                        // and ints
  ...


};


UPInt i;


++i;                                    // calls i.operator++();
i++;                                    // calls i.operator++(0);


--i;                                    // calls i.operator--();
i--;                                    // calls i.operator--(0);
```

This convention is a little on the odd side, but you'll get used to it. More important to get used to, however, is this: the prefix and postfix forms of these operators return *different types* . In particular, prefix forms return a reference, postfix forms return a `const` object. We'll focus here on the prefix and postfix ++ operators, but the story for the -- operators is analogous.

From your days as a C programmer, you may recall that the prefix form of the increment operator is sometimes called "increment and fetch," while the postfix form is often known as "fetch and increment." These two phrases are important to remember, because they all but act as formal

specifications for how prefix and postfix increment should be implemented:

```
// prefix form: increment and fetch
UPInt& UPInt::operator++()
{
  *this += 1;                              // increment
  return *this;                            // fetch
}


// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
  UPInt oldValue = *this;                  // fetch
  ++(*this);        // increment


  return oldValue;                         // return what was
}                                          // fetched
```

Note how the postfix operator makes no use of its parameter. This is typical. The only purpose of the parameter is to distinguish prefix from postfix function invocation. Many compilers issue warnings (see Item E48) if you fail to use named parameters in the body of the function to which they apply, and this can be annoying. To avoid such warnings, a common strategy is to omit names for parameters you don't plan to use; that's what's been done above.

It's clear why postfix increment must return an object (it's returning an old value), but why a `const` object? Imagine that it did not. Then the following would be legal:

```
UPInt i;


i++++;                                      // apply postfix increment
                                            // twice
```

This is the same as

```
i.operator++(0).operator++(0);
```

and it should be clear that the second invocation of `operator++` is being applied to the object returned from the first invocation.

There are two reasons to abhor this. First, it's inconsistent with the behavior of the built-in types. A good rule to follow when designing classes is *when in doubt, do as the `int`s do* , and the `int` s most certainly do not allow double application of postfix increment:

```
int i;


i++++;                                        // error!
```

The second reason is that double application of postfix increment almost never does what clients expect it to. As noted above, the second application of `operator++` in a double increment changes the value of the object returned from the first invocation, *not* the value of the original object. Hence, if

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

were legal, `i` would be incremented only once. This is counterintuitive and confusing (for both `int`s and `UPInt`s), so it's best prohibited.

C++ prohibits it for `int`s, but you must prohibit it yourself for classes you write. The easiest way to do this is to make the return type of postfix increment a `const` object. Then when compilers see

```
i++++;                                  // same as i.operator++(0).operator+
```

they recognize that the `const` object returned from the first call to `operator++` is being used to call `operator++` again. `operator++`, however, is a non-`const` member function, so `const` objects — such as those returned from postfix `operator++` — can't call it.[2] If you've ever wondered if it makes sense to have functions return `const` objects, now you know: sometimes it does, and postfix increment and decrement are examples. (For another example, turn to Item E21.)

If you're the kind who worries about efficiency, you probably broke into a sweat when you first saw the postfix increment function. That function has to create a temporary object for its return value (see Item 19), and the implementation above also creates an explicit temporary object (`oldValue`) that has to be constructed and destructed. The prefix increment function has no such temporaries. This leads to the possibly startling conclusion that, for efficiency reasons alone, clients of `UPInt` should prefer prefix increment to postfix increment unless they really need the behavior of postfix increment. Let us be explicit about this. When dealing with user-defined types, prefix increment should be used whenever possible, because it's inherently more efficient.

Let us make one more observation about the prefix and postfix increment operators. Except for their return values, they do the same thing: they increment a value. That is, they're *supposed* to do the same thing. How can you be sure the behavior of postfix increment is consistent with that of prefix increment? What guarantee do you have that their implementations won't diverge over time, possibly as a result of different programmers maintaining and enhancing them? Unless you've followed the design principle embodied by the code above, you have no such guarantee. That principle is that postfix increment and decrement should be implemented *in terms of* their prefix counterparts. You then need only maintain the prefix versions, because the postfix versions will automatically behave in a consistent fashion.

As you can see, mastering prefix and postfix increment and decrement is easy. Once you know their proper return types and that the postfix operators should be implemented in terms of the prefix operators, there's very little more to learn.

Back to Item 5: Be wary of user-defined conversion functions
Continue to Item 7: Never overload &&, ||, or ,

[2] Alas, it is not uncommon for compilers to fail to enforce this restriction. Before you write programs that rely on it, test your compilers to make sure they behave correctly.
Return

Back to Item 6: Distinguish between prefix and postfix forms of increment and decrement operators
Continue to Item 8: Understand the different meanings of new and delete

Item 7: Never overload &&, ||, or ,.

Like C, C++ employs short-circuit evaluation of boolean expressions. This means that once the truth or falsehood of an expression has been determined, evaluation of the expression ceases, even if some parts of the expression haven't yet been examined. For example, in this case,

```
char *p;


...


if ((p != 0) && (strlen(p) > 10)) ...
```

there is no need to worry about invoking `strlen` on `p` if it's a null pointer, because if the test of `p` against 0 fails, `strlen` will never be called. Similarly, given

```
int rangeCheck(int index)
{
  if ((index < lowerBound) || (index > upperBound)) ...
  ...
}
```

`index` will never be compared to `upperBound` if it's less than `lowerBound`.

This is the behavior that has been drummed into C and C++ programmers since time immemorial, so this is what they expect. Furthermore, they write programs whose correct behavior *depends* on short-circuit evaluation. In the first code fragment above, for example, it is important that `strlen` not be invoked if `p` is a null pointer, because the standard for C++ states (as does the standard for C) that the result of invoking `strlen` on a null pointer is undefined.

C++ allows you to customize the behavior of the `&&` and `||` operators for user-defined types. You do it by overloading the functions `operator&&` and `operator||`, and you can do this at the global scope or on a per-class basis. If you decide to take advantage of this opportunity, however, you must be aware that you are changing the rules of the game quite radically, because you are replacing short-circuit semantics with *function call* semantics. That is, if you overload `operator&&`, what looks to you like this,

```
if (expression1 && expression2) ...
```

looks to compilers like one of these:

```
if (expression1.operator&&(expression2)) ...
                                // when operator&& is a
                                // member function


if (operator&&(expression1, expression2)) ...
                                // when operator&& is a
                                // global function
```

This may not seem like that big a deal, but function call semantics differ from short-circuit semantics in two crucial ways. First, when a function call is made, *all* parameters must be evaluated, so when calling the functions `operator&&` and `operator||`, *both* parameters are evaluated. There is, in other words, no short circuit. Second, the language specification leaves undefined the order of evaluation of parameters to a function call, so there is no way of knowing whether `expression1` or `expression2` will be evaluated first. This stands in stark contrast to short-circuit evaluation, which *always* evaluates its arguments in left-to-right order.

As a result, if you overload `&&` or `||`, there is no way to offer programmers the behavior they both expect and have come to depend on. So don't overload `&&` or `||`.

The situation with the comma operator is similar, but before we delve into that, I'll pause and let you catch the breath you lost when you gasped, "The comma operator? There's a *comma* operator?" There is indeed.

The comma operator is used to form *expressions*, and you're most likely to run across it in the update part of a `for` loop. The following function, for example, is based on one in the second edition of Kernighan's and Ritchie's classic ◦*The C Programming Language* (Prentice-Hall, 1988):

```
// reverse string s in place
void reverse(char s[])
{
  for (int i = 0, j = strlen(s)-1;
       i < j;
       ++i, --j)         // aha! the comma operator!
  {
    int c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}
```

Here, `i` is incremented and `j` is decremented in the final part of the `for` loop. It is convenient to use the comma operator here, because only an expression is valid in the final part of a `for` loop; separate statements to change the values of `i` and `j` would be illegal.

Just as there are rules in C++ defining how `&&` and `||` behave for built-in types, there are rules defining how the comma operator behaves for such types. An expression containing a comma is evaluated by first evaluating the part of the expression to the left of the comma, then evaluating the expression to the right of the comma; the result of the overall comma expression is the value of the expression on the right. So in the final part of the loop above, compilers first evaluate `++i`, then `--j`, and the result of the comma expression is the value returned from `--j`.

Perhaps you're wondering why you need to know this. You need to know because you need to mimic this behavior if you're going to take it upon yourself to write your own comma operator. Unfortunately, you can't perform the requisite mimicry.

If you write `operator,` as a non-member function, you'll never be able to guarantee that the left-hand expression is evaluated before the right-hand expression, because both expressions will be passed as arguments in a function call (to `operator,`). But you have no control over the order in which a function's arguments are evaluated. So the non-member approach is definitely out.

That leaves only the possibility of writing `operator,` as a member function. Even here you can't rely on the left-hand operand to the comma operator being evaluated first, because compilers are not constrained to do things that way. Hence, you can't overload the comma operator and also guarantee it will behave the way it's supposed to. It therefore seems imprudent to overload it at all.

You may be wondering if there's an end to this overloading madness. After all, if you can overload the comma operator, what *can't* you overload? As it turns out, there are limits. You can't overload the following operators:

```
.            .*           ::          ?:
new          delete       sizeof      typeid
static_cast  dynamic_cast  const_cast  reinterpret_cast
```

You can overload these:

```
operator new        operator delete
operator   new[]    operator delete[]
+    -   *   /   %   ^     &   |     ~
!    =   <   >  +=   -=   *=   /=   %=
^=  &=  |=  <<  >>   >>=  <<=  ==   !=
```

```
<=  >=  &&  ||  ++   --    ,   ->*  ->
()  []
```

(For information on the `new` and `delete` operators, as well as `operator new`, `operator delete`, `operator new[]`, and `operator delete[]`, see Item 8.)

Of course, just because you can overload these operators is no reason to run off and do it. The purpose of operator overloading is to make programs easier to read, write, and understand, not to dazzle others with your knowledge that comma is an operator. If you don't have a good reason for overloading an operator, don't overload it. In the case of `&&`, `||`, and `,`, it's difficult to have a good reason, because no matter how hard you try, you can't make them behave the way they're supposed to.

Back to Item 6: Distinguish between prefix and postfix forms of increment and decrement operators
Continue to Item 8: Understand the different meanings of new and delete

Back to Item 7: Never overload &&, ||, or ,
Continue to Exceptions

Item 8: Understand the different meanings of **`new`** and **`delete`**.

It occasionally seems as if people went out of their way to make C++ terminology difficult to understand. Case in point: the difference between the *`new` operator* and *`operator new`*.

When you write code like this,

```
string *ps = new string("Memory Management");
```

the `new` you are using is the `new` operator. This operator is built into the language and, like `sizeof`, you can't change its meaning: it always does the same thing. What it does is twofold. First, it allocates enough memory to hold an object of the type requested. In the example above, it allocates enough memory to hold a `string` object. Second, it calls a constructor to initialize an object in the memory that was allocated. The `new` operator always does those two things; you can't change its behavior in any way.

What you can change is *how* the memory for an object is allocated. The `new` operator calls a function to perform the requisite memory allocation, and you can rewrite or overload that function to change its behavior. The name of the function the `new` operator calls to allocate memory is `operator new`. Honest.

The `operator new` function is usually declared like this:

```
void * operator new(size_t size);
```

The return type is `void*`, because this function returns a pointer to raw, uninitialized memory. (If you like, you can write a version of `operator new` that initializes the memory to some value before returning a pointer to it, but this is not commonly done.) The `size_t` parameter specifies how much memory to allocate. You can overload `operator new` by adding additional parameters, but the first parameter must always be of type `size_t`. (For information on writing `operator new`, consult Items E8-E10.)

You'll probably never want to call `operator new` directly, but on the off chance you do, you'll call it just like any other function:

```
void *rawMemory = operator new(sizeof(string));
```

Here `operator new` will return a pointer to a chunk of memory large enough to hold a `string` object.

Like `malloc`, `operator new`'s only responsibility is to allocate memory. It knows nothing about constructors. All `operator new` understands is memory allocation. It is the job of the `new` operator to take the raw memory that `operator new` returns and transform it into an object. When your compilers see a statement like

```
string *ps = new string("Memory Management");
```

they must generate code that more or less corresponds to this (see Items E8 and E10, as well as the sidebar to my article on counting objects, for a more detailed treatment of this point):

```
void *memory =                              // get raw memory
  operator new(sizeof(string));             // for a string
                                            // object

call string::string("Memory Management")    // initialize the
on *memory;                                 // object in the
                                            // memory

string *ps =                                // make ps point to
  static_cast<string*>(memory);             // the new object
```

Notice that the second step above involves calling a constructor, something you, a mere programmer, are prohibited from doing. Your compilers are unconstrained by mortal limits, however, and they can do whatever they like. That's why you must use the `new` operator if you want to conjure up a heap-based object: you can't directly call the constructor necessary to initialize the object (including such crucial components as its vtbl — see Item 24).

Placement **new**

There are times when you really *want* to call a constructor directly. Invoking a constructor on an existing object makes no sense, because constructors initialize objects, and an object can only be initialized — given its first value — once. But occasionally you have some raw memory that's already been allocated, and you need to construct an object in the memory you have. A special

version of operator new called *placement new* allows you to do it.

As an example of how placement new might be used, consider this:

```
class Widget {
public:
  Widget(int widgetSize);
  ...
};


Widget * constructWidgetInBuffer(void *buffer,
                                 int widgetSize)
{
  return new (buffer) Widget(widgetSize);
}
```

This function returns a pointer to a Widget object that's constructed *within* the buffer passed to the function. Such a function might be useful for applications using shared memory or memory-mapped I/O, because objects in such applications must be placed at specific addresses or in memory allocated by special routines. (For a different example of how placement new can be used, see Item 4.)

Inside constructWidgetInBuffer, the expression being returned is

```
new (buffer) Widget(widgetSize)
```

This looks a little strange at first, but it's just a use of the new operator in which an additional argument (buffer) is being specified for the implicit call that the new operator makes to operator new. The operator new thus called must, in addition to the mandatory size_t argument, accept a void* parameter that points to the memory the object being constructed is to occupy. That operator new *is* placement new, and it looks like this:

```
void * operator new(size_t, void *location)
{
  return location;
}
```

This is probably simpler than you expected, but this is all placement new needs to do. After all, the purpose of operator new is to find memory for an object and return a pointer to that memory. In the case of placement new, the caller already knows what the pointer to the memory should be, because the caller knows where the object is supposed to be placed. All placement new has to do,

then, is return the pointer that's passed into it. (The unused (but mandatory) `size_t` parameter has no name to keep compilers from complaining about its not being used; see Item 6.) Placement `new` is part of the standard C++ library (see Item E49). To use placement `new`, all you have to do is `#include <new>` (or, if your compilers don't yet support the new-style header names (again, see Item E49), `<new.h>`).

If we step back from placement `new` for a moment, we'll see that the relationship between the `new` operator and `operator new`, though you want to create an object on the heap, use the `new` operator. It both allocates memory and calls a constructor for the object. If you only want to allocate memory, call `operator new`; no constructor will be called. If you want to customize the memory allocation that takes place when heap objects are created, write your own version of `operator new` and use the `new` operator; it will automatically invoke your custom version of `operator new`. If you want to construct an object in memory you've already got a pointer to, use placement `new`.

(For additional insights into variants of `new` and `delete`, see my article on counting objects.)

Deletion and Memory Deallocation

To avoid resource leaks, every dynamic allocation must be matched by an equal and opposite deallocation. The function `operator delete` is to the built-in `delete` operator as `operator new` is to the `new` operator. When you say something like this,

```
string *ps;
...
delete ps;                          // use the delete operator
```

your compilers must generate code both to destruct the object `ps` points to and to deallocate the memory occupied by that object.

The memory deallocation is performed by the `operator delete` function, which is usually declared like this:

```
void operator delete(void *memoryToBeDeallocated);
```

Hence,

```
delete ps;
```

causes compilers to generate code that approximately corresponds to this:

```
ps->~string();                      // call the object's dtor



operator delete(ps);                // deallocate the memory
                                    // the object occupied
```

One implication of this is that if you want to deal only with raw, uninitialized memory, you should bypass the `new` and `delete` operators entirely. Instead, you should call `operator new` to get the memory and `operator delete` to return it to the system:

```
void *buffer =                      // allocate enough
  operator new(50*sizeof(char));    // memory to hold 50
                                    // chars; call no ctors


...



operator delete(buffer);            // deallocate the memory;
                                    // call no dtors
```

This is the C++ equivalent of calling `malloc` and `free`.

If you use placement `new` to create an object in some memory, you should avoid using the `delete` operator on that memory. That's because the `delete` operator calls `operator delete` to deallocate the memory, but the memory containing the object wasn't allocated by `operator new` in the first place; placement `new` just returned the pointer that was passed to it. Who knows where that pointer came from? Instead, you should undo the effect of the constructor by explicitly calling the object's destructor:

```
// functions for allocating and deallocating memory in
// shared memory
void * mallocShared(size_t size);
void freeShared(void *memory);



void *sharedMemory = mallocShared(sizeof(Widget));
```

```
Widget *pw =                                        // as above,
  constructWidgetInBuffer(sharedMemory, 10);        // placement
                                                    // new is used


...


delete pw;              // undefined! sharedMemory came from
                        // mallocShared, not operator new


pw->~Widget();          // fine, destructs the Widget pointed to
                        // by pw, but doesn't deallocate the
                        // memory containing the Widget


freeShared(pw);         // fine, deallocates the memory pointed
                        // to by pw, but calls no destructor
```

As this example demonstrates, if the raw memory passed to placement `new` was itself dynamically allocated (through some unconventional means), you must still deallocate that memory if you wish to avoid a memory leak. (See the sidebar to my article on counting objects for information on "placement `delete`".)

Arrays

So far so good, but there's farther to go. Everything we've examined so far concerns itself with only one object at a time. What about array allocation? What happens here?

```
string *ps = new string[10];          // allocate an array of
                                      // objects
```

The `new` being used is still the `new` operator, but because an array is being created, the `new` operator behaves slightly differently from the case of single-object creation. For one thing, memory is no longer allocated by `operator new`. Instead, it's allocated by the array-allocation equivalent, a function called `operator new[]` (often referred to as "array `new`.") Like `operator new`, `operator new[]` can be overloaded. This allows you to seize control of memory allocation for arrays in the same way you can control memory allocation for single objects (but see Item E8 for some caveats on this).

(`operator new[]` is a relatively recent addition to C++, so your compilers may not support it yet. If they don't, the global version of `operator new` will be used to allocate memory for every array, regardless of the type of objects in the array. Customizing array-memory allocation under such compilers is daunting, because it requires that you rewrite the global `operator new`. This is not a task to be undertaken lightly. By default, the global `operator new` handles *all* dynamic memory allocation in a program, so any change in its behavior has a dramatic and pervasive effect. Furthermore, there is only one global `operator new` with the "normal" signature (i.e., taking the single `size_t` parameter — see Item E9), so if you decide to claim it as your own, you instantly render your software incompatible with any library that makes the same decision. (See also Item 27.) As a result of these considerations, custom memory management for arrays is not usually a reasonable design decision for compilers lacking support for `operator new[]`.)

The second way in which the `new` operator behaves differently for arrays than for objects is in the number of constructor calls it makes. For arrays, a constructor must be called for *each object* in the array:

```
string *ps =                // call operator new[] to allocate
  new string[10];           // memory for 10 string objects,
                            // then call the default string
                            // ctor for each array element
```

Similarly, when the `delete` operator is used on an array, it calls a destructor for each array element and then calls `operator delete[]` to deallocate the memory:

```
delete [] ps;               // call the string dtor for each
                            // array element, then call
                            // operator delete[] to
                            // deallocate the array's memory
```

Just as you can replace or overload `operator delete`, you can replace or overload `operator delete[]`. There are some restrictions on how they can be overloaded, however; consult a good C++ text for details. (For ideas on good C++ texts, see the recommendations beginning on page 285.)

So there you have it. The `new` and `delete` operators are built-in and beyond your control, but the memory allocation and deallocation functions they call are not. When you think about customizing the behavior of the `new` and `delete` operators, remember that you can't really do it. You can modify *how* they do what they do, but what they do is fixed by the language.

Back to Item 7: Never overload &&, ||, or ,
Continue to Exceptions

Back to Item 8: Understand the different meanings of new and delete
Continue to Item 9: Use destructors to prevent resource leaks

Exceptions

The addition of exceptions to C++ changes things. Profoundly. Radically. Possibly uncomfortably. The use of raw, unadorned pointers, for example, becomes risky. Opportunities for resource leaks increase in number. It becomes more difficult to write constructors and destructors that behave the way we want them to. Special care must be taken to prevent program execution from abruptly halting. Executables and libraries typically increase in size and decrease in speed.

And these are just the things we know. There is much the C++ community does not know about writing programs using exceptions, including, for the most part, how to do it correctly. There is as yet no agreement on a body of techniques that, when applied routinely, leads to software that behaves predictably and reliably when exceptions are thrown. (For insight into some of the issues involved, see the article by Tom Cargill. For information on recent progress in dealing with these issues, see the articles by Jack Reeves and Herb Sutter.)

We do know this much: programs that behave well in the presence of exceptions do so because they were *designed* to, not because they happen to. Exception-safe programs are not created by accident. The chances of a program behaving well in the presence of exceptions when it was not designed for exceptions are about the same as the chances of a program behaving well in the presence of multiple threads of control when it was not designed for multi-threaded execution: about zero.

That being the case, why use exceptions? Error codes have sufficed for C programmers ever since C was invented, so why mess with exceptions, especially if they're as problematic as I say? The answer is simple: exceptions cannot be ignored. If a function signals an exceptional condition by setting a status variable or returning an error code, there is no way to guarantee the function's caller will check the variable or examine the code. As a result, execution may continue long past the point where the condition was encountered. If the function signals the condition by throwing an exception, however, and that exception is not caught, program execution immediately ceases.

This is behavior that C programmers can approach only by using `setjmp` and `longjmp`. But `longjmp` exhibits a serious deficiency when used with C++: it fails to call destructors for local objects when it adjusts the stack. Most C++ programs depend on such destructors being called, so `setjmp` and `longjmp` make a poor substitute for true exceptions. If you need a way of signaling exceptional conditions that cannot be ignored, and if you must ensure that local destructors are called when searching the stack for code that can handle exceptional conditions, you need C++ exceptions. It's as simple as that.

Because we have much to learn about programming with exceptions, the Items that follow comprise an incomplete guide to writing exception-safe software. Nevertheless, they introduce

important considerations for anyone using exceptions in C++. By heeding the guidance in the material below (and in the magazine articles on this CD), you'll improve the correctness, robustness, and efficiency of the software you write, and you'll sidestep many problems that commonly arise when working with exceptions.

Back to Item 8: Understand the different meanings of new and delete
Continue to Item 9: Use destructors to prevent resource leaks

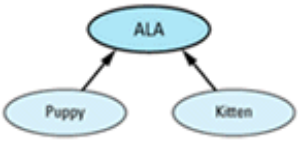
Back to Exceptions
Continue to Item 10: Prevent resource leaks in constructors

Item 9: Use destructors to prevent resource leaks.

Say good-bye to pointers. Admit it: you never really liked them that much anyway.

Okay, you don't have to say good-bye to *all* pointers, but you do need to say *sayonara* to pointers that are used to manipulate local resources. Suppose, for example, you're writing software at the Shelter for Adorable Little Animals, an organization that finds homes for puppies and kittens. Each day the shelter creates a file containing information on the adoptions it arranged that day, and your job is to write a program to read these files and do the appropriate processing for each adoption.

A reasonable approach to this task is to define an abstract base class, `ALA` ("Adorable Little Animal"), plus concrete derived classes for puppies and kittens. A virtual function, `processAdoption`, handles the necessary species-specific processing:




```
class ALA {
public:
  virtual void processAdoption() = 0;
  ...
};


class Puppy: public ALA {
public:
  virtual void processAdoption();
  ...
};


class Kitten: public ALA {
public:
  virtual void processAdoption();
  ...
};
```

You'll need a function that can read information from a file and produce either a `Puppy` object or a `Kitten` object, depending on the information in the file. This is a perfect job for a *virtual constructor*, a kind of function described in Item 25. For our purposes here, the function's declaration is all we need:

```
// read animal information from s, then return a pointer
// to a newly allocated object of the appropriate type
ALA * readALA(istream& s);
```

The heart of your program is likely to be a function that looks something like this:

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {                  // while there's data
    ALA *pa = readALA(dataSource);      // get next animal
    pa->processAdoption();              // process adoption
    delete pa;                          // delete object that
  }                                     // readALA returned
}
```

This function loops through the information in `dataSource`, processing each entry as it goes. The only mildly tricky thing is the need to remember to delete `pa` at the end of each iteration. This is necessary because `readALA` creates a new heap object each time it's called. Without the call to `delete`, the loop would contain a resource leak.

Now consider what would happen if `pa->processAdoption` threw an exception. `processAdoptions` fails to catch exceptions, so the exception would propagate to `processAdoptions`'s caller. In doing so, all statements in `processAdoptions` after the call to `pa->processAdoption` would be skipped, and that means `pa` would never be deleted. As a result, anytime `pa->processAdoption` throws an exception, `processAdoptions` contains a resource leak.

Plugging the leak is easy enough,

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    ALA *pa = readALA(dataSource);



    try {
      pa->processAdoption();
    }
    catch (...) {           // catch all exceptions



      delete pa;            // avoid resource leak when an
                            // exception is thrown
```

```
      throw;                // propagate exception to caller
    }


  delete pa;                // avoid resource leak when no
  }                         // exception is thrown
}
```

but then you have to litter your code with `try` and `catch` blocks. More importantly, you are forced to duplicate cleanup code that is common to both normal and exceptional paths of control. In this case, the call to `delete` must be duplicated. Like all replicated code, this is annoying to write and difficult to maintain, but it also *feels wrong*. Regardless of whether we leave `processAdoptions` by a normal return or by throwing an exception, we need to delete `pa`, so why should we have to say that in more than one place?

We don't have to if we can somehow move the cleanup code that must always be executed into the destructor for an object local to `processAdoptions`. That's because local objects are always destroyed when leaving a function, regardless of how that function is exited. (The only exception to this rule is when you call `longjmp`, and this shortcoming of `longjmp` is the primary reason why C++ has support for exceptions in the first place.) Our real concern, then, is moving the `delete` from `processAdoptions` into a destructor for an object local to `processAdoptions`.

The solution is to replace the pointer `pa` with an object that *acts like* a pointer. That way, when the pointer-like object is (automatically) destroyed, we can have its destructor call `delete`. Objects that act like pointers, but do more, are called *smart pointers*, and, as Item 28 explains, you can make pointer-like objects very smart indeed. In this case, we don't need a particularly brainy pointer, we just need a pointer-like object that knows enough to delete what it points to when the pointer-like object goes out of scope.

It's not difficult to write a class for such objects, but we don't need to. The standard C++ library (see Item E49) contains a class template called `auto_ptr` that does just what we want. Each `auto_ptr` class takes a pointer to a heap object in its constructor and deletes that object in its destructor. Boiled down to these essential functions, `auto_ptr` looks like this:

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *p = 0): ptr(p) {}        // save ptr to object
  ~auto_ptr() { delete ptr; }          // delete ptr to object
```

```
private:
  T *ptr;                                 // raw ptr to object
};
```

The standard version of `auto_ptr` is much fancier, and this stripped-down implementation isn't suitable for real use[3] (we must add at least the copy constructor, assignment operator, and pointer-emulating functions discussed in Item 28), but the concept behind it should be clear: use `auto_ptr` objects instead of raw pointers, and you won't have to worry about heap objects not being deleted, not even when exceptions are thrown. (Because the `auto_ptr` destructor uses the single-object form of `delete`, `auto_ptr` is not suitable for use with pointers to *arrays* of objects. If you'd like an `auto_ptr`-like template for arrays, you'll have to write your own. In such cases, however, it's often a better design decision to use a `vector` instead of an array, anyway.)

Using an `auto_ptr` object instead of a raw pointer, `processAdoptions` looks like this:

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    auto_ptr<ALA> pa(readALA(dataSource));
    pa->processAdoption();
  }
}
```

This version of `processAdoptions` differs from the original in only two ways. First, `pa` is declared to be an `auto_ptr<ALA>` object, not a raw `ALA*` pointer. Second, there is no `delete` statement at the end of the loop. That's it. Everything else is identical, because, except for destruction, `auto_ptr` objects act just like normal pointers. Easy, huh?

The idea behind `auto_ptr` — using an object to store a resource that needs to be automatically released and relying on that object's destructor to release it — applies to more than just pointer-based resources. Consider a function in a GUI application that needs to create a window to display some information:

```
// this function may leak resources if an exception
// is thrown
void displayInfo(const Information& info)
{
  WINDOW_HANDLE w(createWindow());

  display info in window corresponding to w;

  destroyWindow(w);
}
```

Many window systems have C-like interfaces that use functions like `createWindow` and `destroyWindow` to acquire and release window resources. If an exception is thrown during the process of displaying `info` in `w`, the window for which `w` is a handle will be lost just as surely as any other dynamically allocated resource.

The solution is the same as it was before. Create a class whose constructor and destructor acquire and release the resource:

```
// class for acquiring and releasing a window handle
class WindowHandle {
public:
  WindowHandle(WINDOW_HANDLE handle): w(handle) {}
  ~WindowHandle() { destroyWindow(w); }


  operator WINDOW_HANDLE() { return w; }        // see below


private:
  WINDOW_HANDLE w;



  // The following functions are declared private to prevent
  // multiple copies of a WINDOW_HANDLE from being created.
  // See Item 28 for a discussion of a more flexible approach.
  WindowHandle(const WindowHandle&);
  WindowHandle& operator=(const WindowHandle&);
};
```

This looks just like the `auto_ptr` template, except that assignment and copying are explicitly prohibited (see Item E27), and there is an implicit conversion operator that can be used to turn a `WindowHandle` into a `WINDOW_HANDLE`. This capability is essential to the practical application of a `WindowHandle` object, because it means you can use a `WindowHandle` just about anywhere you would normally use a raw `WINDOW_HANDLE`. (See Item 5, however, for why you should generally be leery of implicit type conversion operators.)

Given the `WindowHandle` class, we can rewrite `displayInfo` as follows:

```
// this function avoids leaking resources if an
// exception is thrown
void displayInfo(const Information& info)
{
  WindowHandle w(createWindow());

  display info in window corresponding to w;
```

```
}
```

Even if an exception is thrown within `displayInfo`, the window created by `createWindow` will always be destroyed.

By adhering to the rule that resources should be encapsulated inside objects, you can usually avoid resource leaks in the presence of exceptions. But what happens if an exception is thrown while you're in the process of acquiring a resource, e.g., while you're in the constructor of a resource-acquiring class? What happens if an exception is thrown during the automatic destruction of such resources? Don't constructors and destructors call for special techniques? They do, and you can read about them in Items 10 and 11.

Back to Exceptions
Continue to Item 10: Prevent resource leaks in constructors

---

[3] A complete version of an almost-standard `auto_ptr` appears on pages 291-294.
Return

Back to Item 9: Use destructors to prevent resource leaks
Continue to Item 11: Prevent exceptions from leaving destructors

Item 10: Prevent resource leaks in constructors.

Imagine you're developing software for a multimedia address book. Such an address book might hold, in addition to the usual textual information of a person's name, address, and phone numbers, a picture of the person and the sound of their voice (possibly giving the proper pronunciation of their name).

To implement the book, you might come up with a design like this:

```
class Image {                        // for image data
public:
  Image(const string& imageDataFileName);
  ...
};

class AudioClip {                    // for audio data
public:
  AudioClip(const string& audioDataFileName);
  ...
};

class PhoneNumber {         ... };   // for holding phone numbers


class BookEntry {                    // for each entry in the
public:                              // address book

  BookEntry(const string& name,
            const string& address = "",
            const string& imageFileName = "",
            const string& audioClipFileName = "");
  ~BookEntry();

  // phone numbers are added via this function
  void addPhoneNumber(const PhoneNumber& number);
  ...

private:
  string theName;                 // person's name
  string theAddress;              // their address
  list<PhoneNumber> thePhones;    // their phone numbers
  Image *theImage;                // their image
  AudioClip *theAudioClip;        // an audio clip from them
};
```

Each `BookEntry` must have name data, so you require that as a constructor argument (see Item 3),

but the other fields — the person's address and the names of files containing image and audio data — are optional. Note the use of the `list` class to hold the person's phone numbers. This is one of several container classes that are part of the standard C++ library (see Item E49 and Item 35).

A straightforward way to write the `BookEntry` constructor and destructor is as follows:

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     Const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  if (imageFileName != "") {
    theImage = new Image(imageFileName);
  }

  if (audioClipFileName != "") {
    theAudioClip = new AudioClip(audioClipFileName);
  }
}

BookEntry::~BookEntry()
{
  delete theImage;
  delete theAudioClip;
}
```

The constructor initializes the pointers `theImage` and `theAudioClip` to null, then makes them point to real objects if the corresponding arguments are non-empty strings. The destructor deletes both pointers, thus ensuring that a `BookEntry` object doesn't give rise to a resource leak. Because C++ guarantees it's safe to delete null pointers, `BookEntry`'s destructor need not check to see if the pointers actually point to something before deleting them.

Everything looks fine here, and under normal conditions everything is fine, but under abnormal conditions — under *exceptional* conditions — things are not fine at all.

Consider what will happen if an exception is thrown during execution of this part of the `BookEntry` constructor:

```
if (audioClipFileName != "") {
  theAudioClip = new AudioClip(audioClipFileName);
}
```

An exception might arise because `operator new` (see Item 8) is unable to allocate enough memory

for an `AudioClip` object. One might also arise because the `AudioClip` constructor itself throws an exception. Regardless of the cause of the exception, if one is thrown within the `BookEntry` constructor, it will be propagated to the site where the `BookEntry` object is being created.

Now, if an exception is thrown during creation of the object `theAudioClip` is supposed to point to (thus transferring control out of the `BookEntry` constructor), who deletes the object that `theImage` already points to? The obvious answer is that `BookEntry`'s destructor does, but the obvious answer is wrong. `BookEntry`'s destructor will never be called. Never.

C++ destroys only *fully constructed* objects, and an object isn't fully constructed until its constructor has run to completion. So if a `BookEntry` object `b` is created as a local object,

```
void testBookEntryClass()
{
  BookEntry b("Addison-Wesley Publishing Company",
              "One Jacob Way, Reading, MA 01867");

  ...

}
```

and an exception is thrown during construction of `b`, `b`'s destructor will not be called. Furthermore, if you try to take matters into your own hands by allocating `b` on the heap and then calling `delete` if an exception is thrown,

```
void testBookEntryClass()
{
  BookEntry *pb = 0;

  try {
    pb = new BookEntry("Addison-Wesley Publishing Company",
                       "One Jacob Way, Reading, MA 01867");
    ...
  }
  catch (...) {                 // catch all exceptions

     delete pb;                 // delete pb when an
                                // exception is thrown

     throw;                     // propagate exception to
  }                             // caller

  delete pb;                    // delete pb normally
}
```

you'll find that the `Image` object allocated inside `BookEntry`'s constructor is still lost, because no

assignment is made to `pb` unless the `new` operation succeeds. If `BookEntry`'s constructor throws an exception, `pb` will be the null pointer, so deleting it in the `catch` block does nothing except make you feel better about yourself. Using the smart pointer class `auto_ptr<BookEntry>` (see Item 9) instead of a raw `BookEntry*` won't do you any good either, because the assignment to `pb` still won't be made unless the `new` operation succeeds.

There is a reason why C++ refuses to call destructors for objects that haven't been fully constructed, and it's not simply to make your life more difficult. It's because it would, in many cases, be a nonsensical thing — possibly a harmful thing — to do. If a destructor were invoked on an object that wasn't fully constructed, how would the destructor know what to do? The only way it could know would be if bits had been added to each object indicating how much of the constructor had been executed. Then the destructor could check the bits and (maybe) figure out what actions to take. Such bookkeeping would slow down constructors, and it would make each object larger, too. C++ avoids this overhead, but the price you pay is that partially constructed objects aren't automatically destroyed. (For an example of a similar trade-off involving efficiency and program behavior, turn to Item E13.)

Because C++ won't clean up after objects that throw exceptions during construction, you must design your constructors so that they clean up after themselves. Often, this involves simply catching all possible exceptions, executing some cleanup code, then rethrowing the exception so it continues to propagate. This strategy can be incorporated into the `BookEntry` constructor like this:

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                 const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {                               // this try block is new
    if (imageFileName != "") {
      theImage = new Image(imageFileName);
    }

    if (audioClipFileName != "") {
      theAudioClip = new AudioClip(audioClipFileName);
    }
  }
  catch (...) {                       // catch any exception


    delete theImage;                  // perform necessary
    delete theAudioClip;              // cleanup actions


    throw;                            // propagate the exception
  }
}
```

There is no need to worry about `BookEntry`'s non-pointer data members. Data members are automatically initialized before a class's constructor is called, so if a `BookEntry` constructor body begins executing, the object's `theName`, `theAddress`, and `thePhones` data members have already been fully constructed. As fully constructed objects, these data members will be automatically destroyed when the `BookEntry` object containing them is, and there is no need for you to intervene. Of course, if these objects' constructors call functions that might throw exceptions, *those* constructors have to worry about catching the exceptions and performing any necessary cleanup before allowing them to propagate.

You may have noticed that the statements in `BookEntry`'s `catch` block are almost the same as those in `BookEntry`'s destructor. Code duplication here is no more tolerable than it is anywhere else, so the best way to structure things is to move the common code into a private helper function and have both the constructor and the destructor call it:

```
class BookEntry {
public:
  ...                      // as before

private:
  ...
  void cleanup();          // common cleanup statements
};

void BookEntry::cleanup()
{
  delete theImage;
  delete theAudioClip;
}

BookEntry::BookEntry(const string& name,
                     const string& address,
               const string& imageFileName,
                const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {
    ...                    // as before
  }
  catch (...)   {
    cleanup();             // release resources
    throw;                 // propagate exception
  }
}

BookEntry::~BookEntry()
{
  cleanup();
}
```

This is nice, but it doesn't put the topic to rest. Let us suppose we design our `BookEntry` class slightly differently so that `theImage` and `theAudioClip` are *constant* pointers:

```
class BookEntry {
public:
  ...                                   // as above

private:
  ...
  Image * const theImage;               // pointers are now
  AudioClip * const theAudioClip;       // const
};
```

Such pointers must be initialized via the member initialization lists of `BookEntry`'s constructors, because there is no other way to give `const` pointers a value (see Item E12). A common temptation is to initialize `theImage` and `theAudioClip` like this,

```
// an implementation that may leak resources if an
// exception is thrown
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

but this leads to the problem we originally wanted to eliminate: if an exception is thrown during initialization of `theAudioClip`, the object pointed to by `theImage` is never destroyed. Furthermore, we can't solve the problem by adding `try` and `catch` blocks to the constructor, because `try` and `catch` are statements, and member initialization lists allow only expressions. (That's why we had to use the `?:` syntax instead of the `if-then-else` syntax in the initialization of `theImage` and `theAudioClip`.)

Nevertheless, the only way to perform cleanup chores before exceptions propagate out of a constructor is to catch those exceptions, so if we can't put `try` and `catch` in a member initialization list, we'll have to put them somewhere else. One possibility is inside private member functions that return pointers with which `theImage` and `theAudioClip` should be initialized:

```
class BookEntry {
public:
  ...                     // as above

private:
  ...                     // data members as above

Image * initImage(const string& imageFileName);
  AudioClip * initAudioClip(const string&
                            audioClipFileName);
};

BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(initImage(imageFileName)),
  theAudioClip(initAudioClip(audioClipFileName))
{}

// theImage is initialized first, so there is no need to
// worry about a resource leak if this initialization
// fails. This function therefore handles no exceptions
Image * BookEntry::initImage(const string& imageFileName)
{
  if (imageFileName != "") return new Image(imageFileName);
  else return 0;
}

// theAudioClip is initialized second, so it must make
// sure theImage's resources are released if an exception
// is thrown during initialization of theAudioClip. That's
// why this function uses try...catch.
AudioClip * BookEntry::initAudioClip(const string&
                                     audioClipFileName)
{
  try {
    if (audioClipFileName != "") {
      return new AudioClip(audioClipFileName);
    }
    else return 0;
  }
  catch (...)   {
    delete theImage;
    throw;
  }
}
```

This is perfectly kosher, and it even solves the problem we've been laboring to overcome. The drawback is that code that conceptually belongs in a constructor is now dispersed across several functions, and that's a maintenance headache.

A better solution is to adopt the advice of Item 9 and treat the objects pointed to by `theImage` and `theAudioClip` as resources to be managed by local objects. This solution takes advantage of the facts that both `theImage` and `theAudioClip` are pointers to dynamically allocated objects and that those objects should be deleted when the pointers themselves go away. This is precisely the set of conditions for which the `auto_ptr` classes (see Item 9) were designed. We can therefore change the raw pointer types of `theImage` and `theAudioClip` to their `auto_ptr` equivalents:

```
class BookEntry {
public:
  ...                                          // as above

private:
  ...
  const auto_ptr<Image> theImage;              // these are now
  const auto_ptr<AudioClip> theAudioClip;      // auto_ptr objects
};
```

Doing this makes `BookEntry`'s constructor leak-safe in the presence of exceptions, and it lets us initialize `theImage` and `theAudioClip` using the member initialization list:

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

In this design, if an exception is thrown during initialization of `theAudioClip`, `theImage` is already a fully constructed object, so it will automatically be destroyed, just like `theName`, `theAddress`, and `thePhones`. Furthermore, because `theImage` and `theAudioClip` are now objects, they'll be destroyed automatically when the `BookEntry` object containing them is. Hence there's no need to manually delete what they point to. That simplifies `BookEntry`'s destructor considerably:

```
BookEntry::~BookEntry()
{}                                     // nothing to do!
```

This means you could eliminate `BookEntry`'s destructor entirely.

It all adds up to this: if you replace pointer class members with their corresponding `auto_ptr` objects, you fortify your constructors against resource leaks in the presence of exceptions, you eliminate the need to manually deallocate resources in destructors, and you allow `const` member pointers to be handled in the same graceful fashion as non-`const` pointers.

Dealing with the possibility of exceptions during construction can be tricky, but `auto_ptr` (and `auto_ptr`-like classes) can eliminate most of the drudgery. Their use leaves behind code that's not only easy to understand, it's robust in the face of exceptions, too.

Back to Item 9: Use destructors to prevent resource leaks
Continue to Item 11: Prevent exceptions from leaving destructors

Back to Item 10: Prevent resource leaks in constructors
Continue to Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

Item 11: Prevent exceptions from leaving destructors.

There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly `deleted`. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation.

That being the case, an exception may or may not be active when a destructor is invoked. Regrettably, there is no way to distinguish between these conditions from inside a destructor.[4] As a result, you must write your destructors under the conservative assumption that an exception *is* active, because if control leaves a destructor due to an exception while another exception is active, C++ calls the `terminate` function. That function does just what its name suggests: it terminates execution of your program. Furthermore, it terminates it *immediately*; not even local objects are destroyed.

As an example, consider a `Session` class for monitoring on-line computer sessions, i.e., things that happen from the time you log in through the time you log out. Each `Session` object notes the date and time of its creation and destruction:

```
class Session {
public:
  Session();
  ~Session();
  ...

private:
  static void logCreation(Session *objAddr);
  static void logDestruction(Session *objAddr);
};
```

The functions `logCreation` and `logDestruction` are used to record object creations and destructions, respectively. We might therefore expect that we could code `Session`'s destructor like this:

```
Session::~Session()
{
  logDestruction(this);
}
```

This looks fine, but consider what would happen if `logDestruction` throws an exception. The exception would not be caught in `Session`'s destructor, so it would be propagated to the caller of that destructor. But if the destructor was itself being called because some other exception had been thrown, the `terminate` function would automatically be invoked, and that would stop your program dead in its tracks.

In many cases, this is not what you'll want to have happen. It may be unfortunate that the `Session` object's destruction can't be logged, it might even be a major inconvenience, but is it really so horrific a prospect that the program can't continue running? If not, you'll have to prevent the exception thrown by `logDestruction` from propagating out of `Session`'s destructor. The only way to do that is by using `try` and `catch` blocks. A naive attempt might look like this,

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {
    cerr  << "Unable to log destruction of Session object "
          << "at address "
          << this
          << ".\n";
  }
}
```

but this is probably no safer than our original code. If one of the calls to `operator<<` in the `catch` block results in an exception being thrown, we're back where we started, with an exception leaving the `Session` destructor.

We could always put a `try` block inside the `catch` block, but that seems a bit extreme. Instead, we'll just forget about logging `Session` destructions if `logDestruction` throws an exception:

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {  }
}
```

The `catch` block appears to do nothing, but appearances can be deceiving. That block prevents exceptions thrown from `logDestruction` from propagating beyond `Session`'s destructor. That's all it needs to do. We can now rest easy knowing that if a `Session` object is destroyed as part of stack unwinding, `terminate` will not be called.

There is a second reason why it's bad practice to allow exceptions to propagate out of destructors. If an exception is thrown from a destructor and is not caught there, that destructor won't run to completion. (It will stop at the point where the exception is thrown.) If the destructor doesn't run to completion, it won't do everything it's supposed to do. For example, consider a modified version of the `Session` class where the creation of a session starts a database transaction and the termination of a session ends that transaction:

```
Session::Session()          // to keep things simple,
{                           // this ctor handles no
                            // exceptions
  logCreation(this);
  startTransaction();       // start DB transaction
}

Session::~Session()
{
  logDestruction(this);
  endTransaction();         // end DB transaction
}
```

Here, if `logDestruction` throws an exception, the transaction started in the `Session` constructor will never be ended. In this case, we might be able to reorder the function calls in `Session`'s destructor to eliminate the problem, but if `endTransaction` might throw an exception, we've no choice but to revert to `try` and `catch` blocks.

We thus find ourselves with two good reasons for keeping exceptions from propagating out of destructors. First, it prevents `terminate` from being called during the stack-unwinding part of exception propagation. Second, it helps ensure that destructors always accomplish everything they are supposed to accomplish. Each argument is convincing in its own right, but together, the case is ironclad. (If you're *still* not convinced, turn to Herb Sutter's article; in particular, to the section entitled, "Destructors That Throw and Why They're Evil.)

Back to Item 10: Prevent resource leaks in constructors
Continue to Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

---

[4] Now there is. In July 1995, the ISO/ANSI standardization committee for C++ added a function, `uncaught_exception`, that returns `true` if an exception is active and has not yet been caught. Return

Back to Item 11: Prevent exceptions from leaving destructors
Continue to Item 13: Catch exceptions by reference

Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function.

The syntax for declaring function parameters is almost the same as that for `catch` clauses:

```
class Widget { ... };                // some class; it makes no
                                     // difference what it is



void f1(Widget w);                   // all these functions
void f2(Widget& w);                  // take parameters of
void f3(const Widget& w);            // type Widget, Widget&, or
void f4(Widget *pw);                 // Widget*
void f5(const Widget *pw);



catch (Widget w) ...                 // all these catch clauses
catch (Widget& w)   ...              // catch exceptions of
catch (const Widget& w) ...          // type Widget, Widget&, or
catch (Widget *pw) ...               // Widget*
catch (const Widget *pw) ...
```

You might therefore assume that passing an exception from a `throw` site to a `catch` clause is basically the same as passing an argument from a function call site to the function's parameter. There are some similarities, to be sure, but there are significant differences, too.

Let us begin with a similarity. You can pass both function parameters and exceptions by value, by reference, or by pointer. What *happens* when you pass parameters and exceptions, however, is quite different. This difference grows out of the fact that when you call a function, control eventually returns to the call site (unless the function fails to return), but when you throw an exception, control does *not* return to the `throw` site.

Consider a function that both passes a `Widget` as a parameter and throws a `Widget` as an exception:

```
// function to read the value of a Widget from a stream
istream operator>>(istream& s, Widget& w);



void passAndThrowWidget()
{
```

```
  Widget localWidget;


  cin >> localWidget;          // pass localWidget to operator>>


  throw localWidget;           // throw localWidget as an exception
}
```

When `localWidget` is passed to `operator>>`, no copying is performed. Instead, the reference `w` inside `operator>>` is bound to `localWidget`, and anything done to `w` is really done to `localWidget`. It's a different story when `localWidget` is thrown as an exception. Regardless of whether the exception is caught by value or by reference (it can't be caught by pointer — that would be a type mismatch), a copy of `localWidget` will be made, and it is the *copy* that is passed to the `catch` clause. This must be the case, because `localWidget` will go out of scope once control leaves `passAndThrowWidget`, and when `localWidget` goes out of scope, its destructor will be called. If `localWidget` itself were passed to a `catch` clause, the clause would receive a destructed `Widget`, an ex-`Widget`, a former `Widget`, the carcass of what once was but is no longer a `Widget`. That would not be useful, and that's why C++ specifies that an object thrown as an exception is *always* copied.

This copying occurs even if the object being thrown is not in danger of being destroyed. For example, if `passAndThrowWidget` declares `localWidget` to be static,

```
void passAndThrowWidget()
{
  static Widget localWidget;        // this is now static; it
                                    // will exist until the
                                    // end of the program


  cin >> localWidget;               // this works as before


  throw localWidget;                // a copy of localWidget is
}                                   // still made and thrown
```

a copy of `localWidget` would still be made when the exception was thrown. This means that even if the exception is caught by reference, it is not possible for the `catch` block to modify `localWidget`; it can only modify a *copy* of `localWidget`. This mandatory copying of exception objects helps explain another difference between parameter passing and throwing an exception: the latter is typically much slower than the former (see Item 15).

When an object is copied for use as an exception, the copying is performed by the object's copy

constructor. This copy constructor is the one in the class corresponding to the object's *static* type, not its dynamic type. For example, consider this slightly modified version of `passAndThrowWidget`:

```
class Widget { ... };
class SpecialWidget: public Widget { ... };


void passAndThrowWidget()
{
  SpecialWidget localSpecialWidget;


  ...


  Widget& rw = localSpecialWidget;      // rw refers to a
                                        // SpecialWidget


  throw rw;                             // this throws an
                                        // exception of type
}                                       // Widget!
```

Here a `Widget` exception is thrown, even though `rw` refers to a `SpecialWidget`. That's because `rw`'s static type is `Widget`, not `SpecialWidget`. That `rw` actually refers to a `SpecialWidget` is of no concern to your compilers; all they care about is `rw`'s static type. This behavior may not be what you want, but it's consistent with all other cases in which C++ copies objects. Copying is always based on an object's static type (but see Item 25 for a technique that lets you make copies on the basis of an object's dynamic type).

The fact that exceptions are copies of other objects has an impact on how you propagate exceptions from a `catch` block. Consider these two `catch` blocks, which at first glance appear to do the same thing:

```
catch (Widget& w)                // catch Widget exceptions
{
  ...                            // handle the exception


  throw;                         // rethrow the exception so it
}                                // continues to propagate
```

```
catch (Widget& w)                // catch Widget exceptions
{
  ...                            // handle the exception



  throw w;                       // propagate a copy of the
}                                // caught exception
```

The only difference between these blocks is that the first one rethrows the current exception, while the second one throws a new copy of the current exception. Setting aside the performance cost of the additional copy operation, is there a difference between these approaches?

There is. The first block rethrows the *current* exception, regardless of its type. In particular, if the exception originally thrown was of type `SpecialWidget`, the first block would propagate a `SpecialWidget` exception, even though `w`'s static type is `Widget`. This is because no copy is made when the exception is rethrown. The second `catch` block throws a *new* exception, which will always be of type `Widget`, because that's `w`'s static type. In general, you'll want to use the

```
throw;
```

syntax to rethrow the current exception, because there's no chance that that will change the type of the exception being propagated. Furthermore, it's more efficient, because there's no need to generate a new exception object.

(Incidentally, the copy made for an exception is a *temporary* object. As Item 19 explains, this gives compilers the right to optimize it out of existence. I wouldn't expect your compilers to work that hard, however. Exceptions are supposed to be rare, so it makes little sense for compiler vendors to pour a lot of energy into their optimization.)

Let us examine the three kinds of `catch` clauses that could catch the `Widget` exception thrown by `passAndThrowWidget`. They are:

```
catch (Widget w) ...             // catch exception by value



catch (Widget& w) ...            // catch exception by
                                 // reference



catch (const Widget& w) ...      // catch exception by
                                 // reference-to-const
```

Right away we notice another difference between parameter passing and exception propagation. A thrown object (which, as explained above, is always a temporary) may be caught by simple reference; it need not be caught by reference-to-`const`. Passing a temporary object to a non-`const` reference parameter is not allowed for function calls (see Item 19), but it is for exceptions.

Let us overlook this difference, however, and return to our examination of copying exception objects. We know that when we pass a function argument by value, we make a copy of the passed object (see Item E22), and we store that copy in a function parameter. The same thing happens when we pass an exception by value. Thus, when we declare a `catch` clause like this,

```
catch (Widget w) ...               // catch by value
```

we expect to pay for the creation of *two* copies of the thrown object, one to create the temporary that all exceptions generate, the second to copy that temporary into `w`. Similarly, when we catch an exception by reference,

```
catch (Widget& w) ...              // catch by reference

catch (const Widget& w) ...        // also catch by reference
```

we still expect to pay for the creation of a copy of the exception: the copy that is the temporary. In contrast, when we pass function parameters by reference, no copying takes place. When throwing an exception, then, we expect to construct (and later destruct) one more copy of the thrown object than if we passed the same object to a function.

We have not yet discussed throwing exceptions by pointer, but throw by pointer is equivalent to pass by pointer. Either way, a copy of the *pointer* is passed. About all you need to remember is not to throw a pointer to a local object, because that local object will be destroyed when the exception leaves the local object's scope. The `catch` clause would then be initialized with a pointer to an object that had already been destroyed. This is the behavior the mandatory copying rule is designed to avoid.

The way in which objects are moved from call or `throw` sites to parameters or `catch` clauses is one way in which argument passing differs from exception propagation. A second difference lies in what constitutes a type match between caller or thrower and callee or catcher. Consider the `sqrt` function from the standard math library:

```
double sqrt(double);                // from <cmath> or <math.h>
```

We can determine the square root of an integer like this:

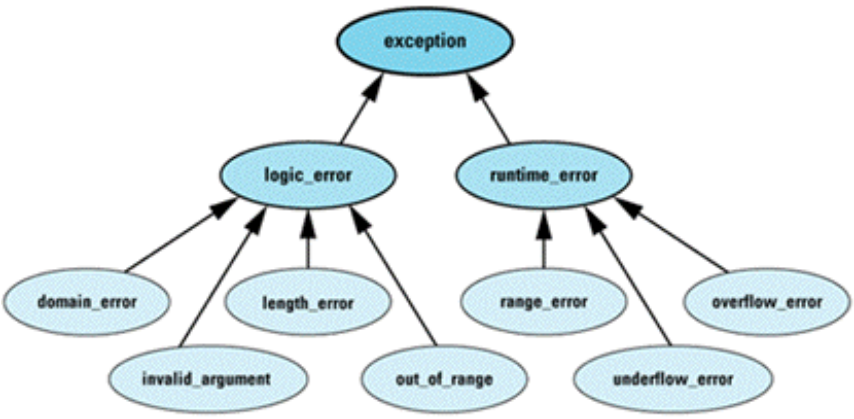
```
int i;

double sqrtOfi = sqrt(i);
```

There is nothing surprising here. The language allows implicit conversion from `int` to `double`, so in the call to `sqrt`, `i` is silently converted to a `double`, and the result of `sqrt` corresponds to that `double`. (See Item 5 for a fuller discussion of implicit type conversions.) In general, such conversions are not applied when matching exceptions to `catch` clauses. In this code,

```
void f(int value)
{
  try {
    if (someFunction()) {      // if someFunction() returns
      throw value;             // true, throw an int
    ...
    }
  }
  catch (double d) {           // handle exceptions of
    ...                        // type double here
  }

  ...

}
```

the `int` exception thrown inside the `try` block will never be caught by the `catch` clause that takes a `double`. That clause catches only exceptions that are exactly of type `double`; no type conversions are applied. As a result, if the `int` exception is to be caught, it will have to be by some other (dynamically enclosing) `catch` clause taking an `int` or an `int&` (possibly modified by `const` or `volatile`).

Two kinds of conversions *are* applied when matching exceptions to `catch` clauses. The first is inheritance-based conversions. A `catch` clause for base class exceptions is allowed to handle exceptions of derived class types, too. For example, consider the diagnostics portion of the hierarchy of exceptions defined by the standard C++ library (see Item E49):



A `catch` clause for `runtime_errors` can catch exceptions of type `range_error` and `overflow_error`, too, and a `catch` clause accepting an object of the root class `exception` can catch any kind of exception derived from this hierarchy.

This inheritance-based exception-conversion rule applies to values, references, and pointers in the usual fashion:

```
catch (runtime_error) ...              // can catch errors of type
catch (runtime_error&) ...             // runtime_error,
catch (const runtime_error&) ...       // range_error, or
                                       // overflow_error
```

```
catch (runtime_error*) ...             // can catch errors of type
catch (const runtime_error*) ...       // runtime_error*,
                                       // range_error*, or
                                       // overflow_error*
```

The second type of allowed conversion is from a typed to an untyped pointer, so a `catch` clause taking a `const void*` pointer will catch an exception of any pointer type:

```
catch (const void*) ...                // catches any exception
```

```
                                        // that's a pointer
```

The final difference between passing a parameter and propagating an exception is that `catch` clauses are always tried *in the order of their appearance*. Hence, it is possible for an exception of a derived class type to be handled by a `catch` clause for one of its base class types — even when a `catch` clause for the derived class is associated with the same `try` block! For example,

```
try {
  ...
}
catch (logic_error& ex) {             // this block will catch
  ...                                 // all logic_error
}                                     // exceptions, even those
                                      // of derived types

catch (invalid_argument& ex) {        // this block can never be
  ...                                 // executed, because all
}                                     // invalid_argument
                                      // exceptions will be caught
                                      // by the clause above
```

Contrast this behavior with what happens when you call a virtual function. When you call a virtual function, the function invoked is the one in the class *closest* to the dynamic type of the object invoking the function. You might say that virtual functions employ a "best fit" algorithm, while exception handling follows a "first fit" strategy. Compilers may warn you if a `catch` clause for a derived class comes after one for a base class (some issue an error, because such code used to be illegal in C++), but your best course of action is preemptive: never put a `catch` clause for a base class before a `catch` clause for a derived class. The code above, for example, should be reordered like this:

```
try {
  ...
}
catch (invalid_argument& ex) {         // handle invalid_argument
  ...                                  // exceptions here
}
catch (logic_error& ex) {              // handle all other
  ...                                  // logic_errors here
}
```

There are thus three primary ways in which passing an object to a function or using that object to invoke a virtual function differs from throwing the object as an exception. First, exception objects are always copied; when caught by value, they are copied twice. Objects passed to function parameters need not be copied at all. Second, objects thrown as exceptions are subject to fewer forms of type conversion than are objects passed to functions. Finally, `catch` clauses are examined

in the order in which they appear in the source code, and the first one that can succeed is selected for execution. When an object is used to invoke a virtual function, the function selected is the one that provides the *best* match for the type of the object, even if it's not the first one listed in the source code.

Back to Item 11: Prevent exceptions from leaving destructors
Continue to Item 13: Catch exceptions by reference

Back to Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function
Continue to Item 14: Use exception specifications judiciously

Item 13: Catch exceptions by reference.

When you write a `catch` clause, you must specify how exception objects are to be passed to that clause. You have three choices, just as when specifying how parameters should be passed to functions: by pointer, by value, or by reference.

Let us consider first catch by pointer. In theory, this should be the least inefficient way to implement the invariably slow process of moving an exception from `throw` site to `catch` clause (see Item 15). That's because throw by pointer is the only way of moving exception information without copying an object (see Item 12). For example:

```
class exception { ... };          // from the standard C++
                                  // library exception
                                  // hierarchy (see Item 12)

void someFunction()
{
  static exception ex;            // exception object

  ...


  throw &ex;                      // throw a pointer to ex

  ...

}

void doSomething()
{
  try {
    someFunction();               // may throw an exception*
  }
  catch (exception *ex) {         // catches the exception*;
    ...                           // no object is copied
  }
}
```

This looks neat and tidy, but it's not quite as well-kept as it appears. For this to work, programmers must define exception objects in a way that guarantees the objects exist after control leaves the functions throwing pointers to them. Global and static objects work fine, but it's easy for programmers to forget the constraint. If they do, they typically end up writing code like this:

```
void someFunction()
{
  exception ex;                  // local exception object;
                                 // will be destroyed when
                                 // this function's scope is
  ...                            // exited


  throw &ex;                     // throw a pointer to an
  ...                            // object that's about to
}                                // be destroyed
```

This is worse than useless, because the `catch` clause handling this exception receives a pointer to an object that no longer exists.

An alternative is to throw a pointer to a new heap object:

```
void someFunction()
{
  ...
  throw new exception;           // throw a pointer to a new heap-
  ...                            // based object (and hope that
}                                // operator new — see Item 8 —
                                 // doesn't itself throw an
                                 // exception!)
```

This avoids the I-just-caught-a-pointer-to-a-destroyed-object problem, but now authors of `catch` clauses confront a nasty question: should they delete the pointer they receive? If the exception object was allocated on the heap, they must, otherwise they suffer a resource leak. If the exception object wasn't allocated on the heap, they mustn't, otherwise they suffer undefined program behavior. What to do?

It's impossible to know. Some clients might pass the address of a global or static object, others might pass the address of an exception on the heap. Catch by pointer thus gives rise to the Hamlet conundrum: to delete or not to delete? It's a question with no good answer. You're best off ducking it.

Furthermore, catch-by-pointer runs contrary to the convention established by the language itself. The four standard exceptions — `bad_alloc` (thrown when `operator new` (see Item 8) can't satisfy a memory request), `bad_cast` (thrown when a `dynamic_cast` to a reference fails; see Item 2), `bad_typeid` (thrown when `dynamic_cast` is applied to a null pointer), and `bad_exception` (available for unexpected exceptions; see Item 14) — are all objects, not pointers to objects, so you have to catch them by value or by reference, anyway.

Catch-by-value eliminates questions about exception deletion and works with the standard exception types. However, it requires that exception objects be copied *twice* each time they're thrown (see Item 12). It also gives rise to the specter of the *slicing problem*, whereby derived class exception objects caught as base class exceptions have their derivedness "sliced off." Such "sliced" objects *are* base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. (Exactly the same thing happens when an object is passed to a function by value — see Item E22.) For example, consider an application employing an exception class hierarchy that extends the standard one:

```
class exception {                   // as above, this is a
public:                             // standard exception class

  virtual const char * what() throw();
                                    // returns a brief descrip.
  ...                               // of the exception (see
                                    // Item 14 for info about
};                                  // the "throw()" at the
                                    // end of the declaration)

class runtime_error:                // also from the standard
  public exception { ... };         // C++ exception hierarchy

class Validation_error:             // this is a class added by
  public runtime_error {            // a client
public:
  virtual const char * what() throw();
                                    // this is a redefinition
  ...                               // of the function declared
};                                  // in class exception above


void someFunction()                 // may throw a validation
{                                   // exception
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}

void doSomething()
{
  try {
    someFunction();                 // may throw a validation
  }                                 // exception


  catch (exception ex) {            // catches all exceptions
                                    // in or derived from
```

```
                                    // the standard hierarchy


    cerr << ex.what();              // calls exception::what(),
    ...                             // never
  }                                 // Validation_error::what()
}
```

The version of `what` that is called is that of the base class, even though the thrown exception is of type `Validation_error` and `Validation_error` redefines that virtual function. This kind of slicing behavior is almost never what you want.

That leaves only catch-by-reference. Catch-by-reference suffers from none of the problems we have discussed. Unlike catch-by-pointer, the question of object deletion fails to arise, and there is no difficulty in catching the standard exception types. Unlike catch-by-value, there is no slicing problem, and exception objects are copied only once.

If we rewrite the last example using catch-by-reference, it looks like this:

```
void someFunction()                 // nothing changes in this
{                                   // function
  ...

  if (a validation test fails) {
    throw Validation_error();
  }

  ...

}

void doSomething()
{
  try {
    someFunction();                 // no change here
  }
  catch (exception& ex) {           // here we catch by reference
                                    // instead of by value


    cerr << ex.what();              // now calls
                                    // Validation_error::what(),
    ...                             // not exception::what()
  }
}
```

There is no change at the `throw` site, and the only change in the `catch` clause is the addition of an ampersand. This tiny modification makes a big difference, however, because virtual functions in the `catch` block now work as we expect: functions in `Validation_error` are invoked if they redefine those in `exception`.

What a happy confluence of events! If you catch by reference, you sidestep questions about object deletion that leave you damned if you do and damned if you don't; you avoid slicing exception objects; you retain the ability to catch standard exceptions; and you limit the number of times exception objects need to be copied. So what are you waiting for? Catch exceptions by reference!

Back to Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function
Continue to Item 14: Use exception specifications judiciously

Back to Item 13: Catch exceptions by reference
Continue to Item 15: Understand the costs of exception handling

Item 14: Use exception specifications judiciously.

There's no denying it: exception specifications have appeal. They make code easier to understand, because they explicitly state what exceptions a function may throw. But they're more than just fancy comments. Compilers are sometimes able to detect inconsistent exception specifications during compilation. Furthermore, if a function throws an exception not listed in its exception specification, that fault is detected at runtime, and the special function `unexpected` is automatically invoked. Both as a documentation aid and as an enforcement mechanism for constraints on exception usage, then, exception specifications seem attractive.

As is often the case, however, beauty is only skin deep. The default behavior for `unexpected` is to call `terminate`, and the default behavior for `terminate` is to call `abort`, so the default behavior for a program with a violated exception specification is to halt. Local variables in active stack frames are not destroyed, because `abort` shuts down program execution without performing such cleanup. A violated exception specification is therefore a cataclysmic thing, something that should almost never happen.

Unfortunately, it's easy to write functions that make this terrible thing occur. Compilers only *partially* check exception usage for consistency with exception specifications. What they do not check for — what the language standard *prohibits* them from rejecting (though they may issue a warning) — is a call to a function that *might* violate the exception specification of the function making the call.

Consider a declaration for a function `f1` that has no exception specification. Such a function may throw any kind of exception:

```
extern void f1();                   // might throw anything
```

Now consider a function `f2` that claims, through its exception specification, it will throw only exceptions of type `int`:

```
void f2() throw(int);
```

It is perfectly legal C++ for `f2` to call `f1`, even though `f1` might throw an exception that would violate `f2`'s exception specification:

```
void f2() throw(int)
{
  ...
  f1();                 // legal even though f1 might throw
                        // something besides an int
```

```
  ...
}
```

This kind of flexibility is essential if new code with exception specifications is to be integrated with older code lacking such specifications.

Because your compilers are content to let you call functions whose exception specifications are inconsistent with those of the routine containing the calls, and because such calls might result in your program's execution being terminated, it's important to write your software in such a way that these kinds of inconsistencies are minimized. A good way to start is to avoid putting exception specifications on templates that take type arguments. Consider this template, which certainly looks as if it couldn't throw any exceptions:

```
// a poorly designed template wrt exception specifications
template<class T>
bool operator==(const T& lhs, const T& rhs) throw()
{
  return &lhs == &rhs;
}
```

This template defines an `operator==` function for all types. For any pair of objects of the same type, it returns `true` if the objects have the same address, otherwise it returns `false`.

This template contains an exception specification stating that the functions generated from the template will throw no exceptions. But that's not necessarily true, because it's possible that `operator&` (the address-of operator — see Item E45) has been overloaded for some types. If it has, `operator&` may throw an exception when called from inside `operator==`. If it does, our exception specification is violated, and off to `unexpected` we go.

This is a specific example of a more general problem, namely, that there is no way to know *anything* about the exceptions thrown by a template's type parameters. We can almost never provide a meaningful exception specification for a template, because templates almost invariably use their type parameter in some way. The conclusion? Templates and exception specifications don't mix.

A second technique you can use to avoid calls to `unexpected` is to omit exception specifications on functions making calls to functions that themselves lack exception specifications. This is simple common sense, but there is one case that is easy to forget. That's when allowing users to register callback functions:

```
// Function pointer type for a window system callback
// when a window system event occurs
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack);
```

```
// Window system class for holding onto callback
// functions registered by window system clients
class CallBack {
public:
  CallBack(CallBackPtr fPtr, void *dataToPassBack)
  : func(fPtr), data(dataToPassBack) {}


  void makeCallBack(int eventXLocation,
                    int eventYLocation) const throw();


private:
  CallBackPtr func;               // function to call when
                                  // callback is made



  void *data;                     // data to pass to callback
};                                // function


// To implement the callback, we call the registered func-
// tion with event's coordinates and the registered data
void CallBack::makeCallBack(int eventXLocation,
                            int eventYLocation) const throw()
{
  func(eventXLocation, eventYLocation, data);
}
```

Here the call to `func` in `makeCallBack` runs the risk of a violated exception specification, because there is no way of knowing what exceptions `func` might throw.

This problem can be eliminated by tightening the exception specification in the `CallBackPtr` typedef:[5]

```
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack) throw();
```

Given this typedef, it is now an error to register a `callback` function that fails to guarantee it throws nothing:

```
// a callback function without an exception specification
void callBackFcn1(int eventXLocation, int eventYLocation,
                  void *dataToPassBack);
```

```
void *callBackData;


...



CallBack c1(callBackFcn1, callBackData);
                                // error! callBackFcn1
                                // might throw an exception


// a callback function with an exception specification
void callBackFcn2(int eventXLocation,
                  int eventYLocation,
                  void *dataToPassBack) throw();



CallBack c2(callBackFcn2, callBackData);
                                // okay, callBackFcn2 has a
                                // conforming ex. spec.
```

This checking of exception specifications when passing function pointers is a relatively recent addition to the language, so don't be surprised if your compilers don't yet support it. If they don't, it's up to you to ensure you don't make this kind of mistake.

A third technique you can use to avoid calls to `unexpected` is to handle exceptions "the system" may throw. Of these exceptions, the most common is `bad_alloc`, which is thrown by `operator new` and `operator new[]` when a memory allocation fails (see Item 8). If you use the `new` operator (again, see Item 8) in any function, you must be prepared for the possibility that the function will encounter a `bad_alloc` exception.

Now, an ounce of prevention may be better than a pound of cure, but sometimes prevention is hard and cure is easy. That is, sometimes it's easier to cope with unexpected exceptions directly than to prevent them from arising in the first place. If, for example, you're writing software that uses exception specifications rigorously, but you're forced to call functions in libraries that don't use exception specifications, it's impractical to prevent unexpected exceptions from arising, because that would require changing the code in the libraries.

If preventing unexpected exceptions isn't practical, you can exploit the fact that C++ allows you to replace unexpected exceptions with exceptions of a different type. For example, suppose you'd like all unexpected exceptions to be replaced by `UnexpectedException` objects. You can set it up like this,

```
class UnexpectedException {};          // all unexpected exception
                                       // objects will be replaced
                                       // by objects of this type


void convertUnexpected()               // function to call if
{                                      // an unexpected exception
  throw UnexpectedException();         // is thrown
}
```

and make it happen by replacing the default `unexpected` function with `convertUnexpected`:

```
set_unexpected(convertUnexpected);
```

Once you've done this, any unexpected exception results in `convertUnexpected` being called. The unexpected exception is then replaced by a new exception of type `UnexpectedException`. Provided the exception specification that was violated includes `UnexpectedException`, exception propagation will then continue as if the exception specification had always been satisfied. (If the exception specification does not include `UnexpectedException`, `terminate` will be called, just as if you had never replaced `unexpected`.)

Another way to translate unexpected exceptions into a well known type is to rely on the fact that if the `unexpected` function's replacement rethrows the current exception, that exception will be replaced by a new exception of the standard type `bad_exception`. Here's how you'd arrange for that to happen:

```
void convertUnexpected()          // function to call if
{                                 // an unexpected exception
  throw;                          // is thrown; just rethrow
}                                 // the current exception

set_unexpected(convertUnexpected);
                                 // install convertUnexpected
                                 // as the unexpected
                                 // replacement
```

If you do this and you include `bad_exception` (or its base class, the standard class `exception`) in all your exception specifications, you'll never have to worry about your program halting if an unexpected exception is encountered. Instead, any wayward exception will be replaced by a `bad_exception`, and that exception will be propagated in the stead of the original one.

By now you understand that exception specifications can be a lot of trouble. Compilers perform only partial checks for their consistent usage, they're problematic in templates, they're easy to violate inadvertently, and, by default, they lead to abrupt program termination when they're violated. Exception specifications have another drawback, too, and that's that they result in `unexpected` being invoked even when a higher-level caller is prepared to cope with the exception that's arisen. For example, consider this code, which is taken almost verbatim from Item 11:

```
class Session {                  // for modeling online
public:                          // sessions
  ~Session();
  ...

private:
  static void logDestruction(Session *objAddr) throw();
};

Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {  }
}
```

The `Session` destructor calls `logDestruction` to record the fact that a `Session` object is being destroyed, but it explicitly catches any exceptions that might be thrown by `logDestruction`. However, `logDestruction` comes with an exception specification asserting that it throws no exceptions. Now, suppose some function called by `logDestruction` throws an exception that `logDestruction` fails to catch. This isn't supposed to happen, but as we've seen, it isn't difficult to write code that leads to the violation of exception specifications. When this unanticipated exception propagates through `logDestruction`, `unexpected` will be called, and, by default, that will result in termination of the program. This is correct behavior, to be sure, but is it the behavior the author of `Session`'s destructor wanted? That author took pains to handle *all possible* exceptions, so it seems almost unfair to halt the program without giving `Session`'s destructor's `catch` block a chance to work. If `logDestruction` had no exception specification, this I'm-willing-to-catch-it-if-you'll-just-give-me-a-chance scenario would never arise. (One way to prevent it is to replace `unexpected` as described above.)

It's important to keep a balanced view of exception specifications. They provide excellent documentation on the kinds of exceptions a function is expected to throw, and for situations in which violating an exception specification is so dire as to justify immediate program termination, they offer that behavior by default. At the same time, they are only partly checked by compilers and they are easy to violate inadvertently. Furthermore, they can prevent high-level exception handlers from dealing with unexpected exceptions, even when they know how to. That being the case, exception specifications are a tool to be applied judiciously. Before adding them to your functions,

consider whether the behavior they impart to your software is really the behavior you want.

Back to Item 13: Catch exceptions by reference
Continue to Item 15: Understand the costs of exception handling

---

[5] Alas, it can't, at least not portably. Though many compilers accept the code shown on this page, the ◦standardization committee has inexplicably decreed that "an exception specification shall not appear in a `typedef`." I don't know why. If you need a portable solution, you must — it hurts me to write this — make `CallBackPtr` a macro, sigh.
Return

Back to Item 14: Use exception specifications judiciously
Continue to Efficiency

Item 15: Understand the costs of exception handling.

To handle exceptions at runtime, programs must do a fair amount of bookkeeping. At each point during execution, they must be able to identify the objects that require destruction if an exception is thrown; they must make note of each entry to and exit from a `try` block; and for each `try` block, they must keep track of the associated `catch` clauses and the types of exceptions those clauses can handle. This bookkeeping is not free. Nor are the runtime comparisons necessary to ensure that exception specifications are satisfied. Nor is the work expended to destroy the appropriate objects and find the correct `catch` clause when an exception is thrown. No, exception handling has costs, and you pay at least some of them even if you never use the keywords `try`, `throw`, or `catch`.

Let us begin with the things you pay for even if you never use any exception-handling features. You pay for the space used by the data structures needed to keep track of which objects are fully constructed (see Item 10), and you pay for the time needed to keep these data structures up to date. These costs are typically quite modest. Nevertheless, programs compiled without support for exceptions are typically both faster and smaller than their counterparts compiled with support for exceptions.

In theory, you don't have a choice about these costs: exceptions are part of C++, compilers have to support them, and that's that. You can't even expect compiler vendors to eliminate the costs if you use no exception-handling features, because programs are typically composed of multiple independently generated object files, and just because one object file doesn't do anything with exceptions doesn't mean others don't. Furthermore, even if none of the object files linked to form an executable use exceptions, what about the libraries they're linked with? If *any part* of a program uses exceptions, the rest of the program must support them, too. Otherwise it may not be possible to provide correct exception-handling behavior at runtime.

That's the theory. In practice, most vendors who support exception handling allow you to control whether support for exceptions is included in the code they generate. If you know that no part of your program uses `try`, `throw`, or `catch`, and you also know that no library with which you'll link uses `try`, `throw`, or `catch`, you might as well compile without exception-handling support and save yourself the size and speed penalty you'd otherwise probably be assessed for a feature you're not using. As time goes on and libraries employing exceptions become more common, this strategy will become less tenable, but given the current state of C++ software development, compiling without support for exceptions is a reasonable performance optimization if you have already decided not to use exceptions. It may also be an attractive optimization for libraries that eschew exceptions, provided they can guarantee that exceptions thrown from client code never propagate into the library. This is a difficult guarantee to make, as it precludes client redefinitions of library-declared virtual functions; it also rules out client-defined callback functions.

A second cost of exception-handling arises from `try` blocks, and you pay it whenever you use one, i.e., whenever you decide you want to be able to catch exceptions. Different compilers implement `try` blocks in different ways, so the cost varies from compiler to compiler. As a rough estimate, expect your overall code size to increase by 5-10% and your runtime to go up by a similar amount if you use `try` blocks. This assumes no exceptions are thrown; what we're discussing here is just the cost of *having* `try` blocks in your programs. To minimize this cost, you should avoid unnecessary `try` blocks.

Compilers tend to generate code for exception specifications much as they do for `try` blocks, so an exception specification generally incurs about the same cost as a `try` block. Excuse me? You say you thought exception specifications were just specifications, you didn't think they generated code? Well, now you have something new to think about.

Which brings us to the heart of the matter, the cost of throwing an exception. In truth, this shouldn't be much of a concern, because exceptions should be rare. After all, they indicate the occurrence of events that are *exceptional*. The 80-20 rule (see Item 16) tells us that such events should almost never have much impact on a program's overall performance. Nevertheless, I know you're curious about just how big a hit you'll take if you throw an exception, and the answer is it's probably a big one. Compared to a normal function return, returning from a function by throwing an exception may be as much as *three orders of magnitude* slower. That's quite a hit. But you'll take it only if you throw an exception, and that should be almost never. If, however, you've been thinking of using exceptions to indicate relatively common conditions like the completion of a data structure traversal or the termination of a loop, now would be an excellent time to think again.

But wait. How can I know this stuff? If support for exceptions is a relatively recent addition to most compilers (it is), and if different compilers implement their support in different ways (they do), how can I say that a program's size will generally grow by about 5-10%, its speed will decrease by a similar amount, and it may run orders of magnitude slower if lots of exceptions are thrown? The answer is frightening: a little rumor and a handful of benchmarks (see Item 23). The fact is that most people — including most compiler vendors — have little experience with exceptions, so though we know there are costs associated with them, it is difficult to predict those costs accurately.

The prudent course of action is to be aware of the costs described in this item, but not to take the numbers very seriously. Whatever the cost of exception handling, you don't want to pay any more than you have to. To minimize your exception-related costs, compile without support for exceptions when that is feasible; limit your use of `try` blocks and exception specifications to those locations where you honestly need them; and throw exceptions only under conditions that are truly exceptional. If you still have performance problems, profile your software (see Item 16) to determine if exception support is a contributing factor. If it is, consider switching to different compilers, ones that provide more efficient implementations of C++'s exception-handling features.

Back to Item 14: Use exception specifications judiciously
Continue to Efficiency

Back to Item 15:Understand the costs of exception handling
Continue to Item 16: Remember the 80-20 rule

Efficiency

I harbor a suspicion that someone has performed secret ◦Pavlovian experiments on C++ software developers. How else can one explain the fact that when the word "efficiency" is mentioned, scores of programmers start to drool?

In fact, efficiency is no laughing matter. Programs that are too big or too slow fail to find acceptance, no matter how compelling their merits. This is perhaps as it should be. Software is supposed to help us do things better, and it's difficult to argue that slower is better, that demanding 32 megabytes of memory is better than requiring a mere 16, that chewing up 100 megabytes of disk space is better than swallowing only 50. Furthermore, though some programs take longer and use more memory because they perform more ambitious computations, too many programs can blame their sorry pace and bloated footprint on nothing more than bad design and slipshod programming.

Writing efficient programs in C++ starts with the recognition that C++ may well have nothing to do with any performance problems you've been having. If you want to write an efficient C++ program, you must first be able to write an efficient *program*. Too many developers overlook this simple truth. Yes, loops may be unrolled by hand and multiplications may be replaced by shift operations, but such micro-tuning leads nowhere if the higher-level algorithms you employ are inherently inefficient. Do you use quadratic algorithms when linear ones are available? Do you compute the same value over and over? Do you squander opportunities to reduce the average cost of expensive operations? If so, you can hardly be surprised if your programs are described like second-rate tourist attractions: worth a look, but only if you've got some extra time.

The material in this chapter attacks the topic of efficiency from two angles. The first is language-independent, focusing on things you can do in any programming language. C++ provides a particularly appealing implementation medium for these ideas, because its strong support for encapsulation makes it possible to replace inefficient class implementations with better algorithms and data structures that support the same interface.

The second focus is on C++ itself. High-performance algorithms and data structures are great, but sloppy implementation practices can reduce their effectiveness considerably. The most insidious mistake is both simple to make and hard to recognize: creating and destroying too many objects. Superfluous object constructions and destructions act like a hemorrhage on your program's performance, with precious clock-ticks bleeding away each time an unnecessary object is created and destroyed. This problem is so pervasive in C++ programs, I devote four separate items to describing where these objects come from and how you can eliminate them without compromising the correctness of your code.

Programs don't get big and slow only by creating too many objects. Other potholes on the road to high performance include library selection and implementations of language features. In the items that follow, I address these issues, too.

After reading the material in this chapter, you'll be familiar with several principles that can improve the performance of virtually any program you write, you'll know exactly how to prevent unnecessary objects from creeping into your software, and you'll have a keener awareness of how your compilers behave when generating executables.

It's been said that forewarned is forearmed. If so, think of the information that follows as preparation for battle.

Back to Item 15: Understand the costs of exception handling
Continue to Item 16: Remember the 80-20 rule

Back to Efficiency
Continue to Item 17: Consider using lazy evaluation

Item 16: Remember the 80-20 rule.

The 80-20 rule states that 80 percent of a program's resources are used by about 20 percent of the code: 80 percent of the runtime is spent in approximately 20 percent of the code; 80 percent of the memory is used by some 20 percent of the code; 80 percent of the disk accesses are performed for about 20 percent of the code; 80 percent of the maintenance effort is devoted to around 20 percent of the code. The rule has been repeatedly verified through examinations of countless machines, operating systems, and applications. The 80-20 rule is more than just a catchy phrase; it's a guideline about system performance that has both wide applicability and a solid empirical basis.

When considering the 80-20 rule, it's important not to get too hung up on numbers. Some people favor the more stringent 90-10 rule, and there's experimental evidence to back that, too. Whatever the precise numbers, the fundamental point is this: the overall performance of your software is almost always determined by a small part of its constituent code.

As a programmer striving to maximize your software's performance, the 80-20 rule both simplifies and complicates your life. On one hand, the 80-20 rule implies that most of the time you can produce code whose performance is, frankly, rather mediocre, because 80 percent of the time its efficiency doesn't affect the overall performance of the system you're working on. That may not do much for your ego, but it should reduce your stress level a little. On the other hand, the rule implies that if your software has a performance problem, you've got a tough job ahead of you, because you not only have to locate the small pockets of code that are causing the problem, you have to find ways to increase their performance dramatically. Of these tasks, the more troublesome is generally locating the bottlenecks. There are two fundamentally different ways to approach the matter: the way most people do it and the right way.

The way most people locate bottlenecks is to guess. Using experience, intuition, tarot cards and Ouija boards, rumors or worse, developer after developer solemnly proclaims that a program's efficiency problems can be traced to network delays, improperly tuned memory allocators, compilers that don't optimize aggressively enough, or some bonehead manager's refusal to permit assembly language for crucial inner loops. Such assessments are generally delivered with a condescending sneer, and usually both the sneerers and their prognostications are flat-out wrong.

Most programmers have lousy intuition about the performance characteristics of their programs, because program performance characteristics tend to be highly unintuitive. As a result, untold effort is poured into improving the efficiency of parts of programs that will never have a noticeable effect on their overall behavior. For example, fancy algorithms and data structures that minimize computation may be added to a program, but it's all for naught if the program is I/O-bound. Souped-up I/O libraries (see Item 23) may be substituted for the ones shipped with compilers, but there's not

much point if the programs using them are CPU-bound.

That being the case, what do you do if you're faced with a slow program or one that uses too much memory? The 80-20 rule means that improving random parts of the program is unlikely to help very much. The fact that programs tend to have unintuitive performance characteristics means that trying to guess the causes of performance bottlenecks is unlikely to be much better than just improving random parts of your program. What, then, *will* work?

What will work is to empirically identify the 20 percent of your program that is causing you heartache, and the way to identify that horrid 20 percent is to use a program profiler. Not just any profiler will do, however. You want one that *directly* measures the resources you are interested in. For example, if your program is too slow, you want a profiler that tells you how much *time* is being spent in different parts of the program. That way you can focus on those places where a significant improvement in local efficiency will also yield a significant improvement in overall efficiency.

Profilers that tell you how many times each statement is executed or how many times each function is called are of limited utility. From a performance point of view, you do not care how many times a statement is executed or a function is called. It is, after all, rather rare to encounter a user of a program or a client of a library who complains that too many statements are being executed or too many functions are being called. If your software is fast enough, nobody cares how many statements are executed, and if it's too slow, nobody cares how few. All they care about is that they hate to wait, and if your program is making them do it, they hate you, too.

Still, knowing how often statements are executed or functions are called can sometimes yield insight into what your software is doing. If, for example, you think you're creating about a hundred objects of a particular type, it would certainly be worthwhile to discover that you're calling constructors in that class thousands of times. Furthermore, statement and function call counts can indirectly help you understand facets of your software's behavior you can't directly measure. If you have no direct way of measuring dynamic memory usage, for example, it may be helpful to know at least how often memory allocation and deallocation functions (e.g., operators `new`, `new[]`, `delete`, and `delete[]` — see Item 8) are called.

Of course, even the best of profilers is hostage to the data it's given to process. If you profile your program while it's processing unrepresentative input data, you're in no position to complain if the profiler leads you to fine-tune parts of your software — the parts making up some 80 percent of it — that have no bearing on its usual performance. Remember that a profiler can only tell you how a program behaved on a particular run (or set of runs), so if you profile a program using input data that is unrepresentative, you're going to get back a profile that is equally unrepresentative. That, in turn, is likely to lead to you to optimize your software's behavior for uncommon uses, and the overall impact on common uses may even be negative.

The best way to guard against these kinds of pathological results is to profile your software using as

many data sets as possible. Moreover, you must ensure that each data set is representative of how the software is used by its clients (or at least its most important clients). It is usually easy to acquire representative data sets, because many clients are happy to let you use their data when profiling. After all, you'll then be tuning your software to meet their needs, and that can only be good for both of you.

Back to Efficiency
Continue to Item 17: Consider using lazy evaluation

Item 17: Consider using lazy evaluation.

From the perspective of efficiency, the best computations are those you never perform at all. That's fine, but if you don't need to do something, why would you put code in your program to do it in the first place? And if you do need to do something, how can you possibly avoid executing the code that does it?

The key is to be lazy.

Remember when you were a child and your parents told you to clean your room? If you were anything like me, you'd say "Okay," then promptly go back to what you were doing. You would *not* clean your room. In fact, cleaning your room would be the last thing on your mind — *until* you heard your parents coming down the hall to confirm that your room had, in fact, been cleaned. Then you'd sprint to your room and get to work as fast as you possibly could. If you were lucky, your parents would never check, and you'd avoid all the work cleaning your room normally entails.

It turns out that the same delay tactics that work for a five year old work for a C++ programmer. In Computer Science, however, we dignify such procrastination with the name *lazy evaluation* . When you employ lazy evaluation, you write your classes in such a way that they defer computations until the *results* of those computations are required. If the results are never required, the computations are never performed, and neither your software's clients nor your parents are any the wiser.

Perhaps you're wondering exactly what I'm talking about. Perhaps an example would help. Well, lazy evaluation is applicable in an enormous variety of application areas, so I'll describe four.

Reference Counting

Consider this code:

```
class String { ... };                          // a string class (the standard
                                               // string type may be implemented
                                               // as described below, but it
                                               // doesn't have to be)

String s1 = "Hello";

String s2 = s1;                                // call String copy ctor
```

A common implementation for the `String` copy constructor would result in `s1` and `s2` each having its own copy of `"Hello"` after `s2` is initialized with `s1` . Such a copy constructor would incur a relatively large expense, because it would have to make a copy of `s1` 's value to give to `s2` , and that would typically entail allocating heap memory via the `new` operator (see Item 8) and calling `strcpy` to copy the data in `s1` into the memory allocated by `s2` . This is *eager evaluation:* making a copy of `s1` and putting it into `s2` just because the `String` copy constructor was *called* . At this point,

however, there has been no real *need* for `s2` to have a copy of the value, because `s2` hasn't been used yet.

The lazy approach is a lot less work. Instead of giving `s2` a copy of `s1` 's value, we have `s2` *share* `s1` 's value. All we have to do is a little bookkeeping so we know who's sharing what, and in return we save the cost of a call to `new` and the expense of copying anything. The fact that `s1` and `s2` are sharing a data structure is transparent to clients, and it certainly makes no difference in statements like the following, because they only read values, they don't write them:

```
cout << s1;                             // read s1's value

cout << s1 + s2;                        // read s1's and s2's values
```

In fact, the only time the sharing of values makes a difference is when one or the other string is *modified* ; then it's important that only one string be changed, not both. In this statement,

```
s2.convertToUpperCase();
```

it's crucial that only `s2` 's value be changed, not `s1` 's also.

To handle statements like this, we have to implement `String` 's `convertToUpperCase` function so that it makes a copy of `s2` 's value and makes that value private to `s2` before modifying it. Inside `convertToUpperCase` , we can be lazy no longer: we have to make a copy of `s2` 's (shared) value for `s2` 's private use. On the other hand, if `s2` is never modified, we never have to make a private copy of its value. It can continue to share a value as long as it exists. If we're lucky, `s2` will never be modified, in which case we'll never have to expend the effort to give it its own value.

The details on making this kind of value sharing work (including all the code) are provided in Item 29 , but the idea is lazy evaluation: don't bother to make a copy of something until you really need one. Instead, be lazy — use someone else's copy as long as you can get away with it. In some application areas, you can often get away with it forever.

Distinguishing Reads from Writes

Pursuing the example of reference-counting strings a bit further, we come upon a second way in which lazy evaluation can help us. Consider this code:

```
String s = "Homer's Iliad";                 // Assume s is a
                                            // reference-counted string
...

cout << s[3];                           // call operator[] to read s[3]
s[3] = 'x';                             // call operator[] to write s[3]
```

The first call to `operator[]` is to read part of a string, but the second call is to perform a write. We'd like to be able to distinguish the read call from the write, because reading a reference-counted

string is cheap, but writing to such a string may require splitting off a new copy of the string's value prior to the write.

This puts us in a difficult implementation position. To achieve what we want, we need to do different things inside `operator[]` (depending on whether it's being called to perform a read or a write). How can we determine whether `operator[]` has been called in a read or a write context? The brutal truth is that we can't. By using lazy evaluation and proxy classes as described in Item 30 , however, we can defer the decision on whether to take read actions or write actions until we can determine which is correct.

Lazy Fetching

As a third example of lazy evaluation, imagine you've got a program that uses large objects containing many constituent fields. Such objects must persist across program runs, so they're stored in a database. Each object has a unique object identifier that can be used to retrieve the object from the database:

```
class LargeObject {                          // large persistent objects
public:
  LargeObject(ObjectID id);                  // restore object from disk

  const string& field1() const;              // value of field 1
  int field2() const;                        // value of field 2
  double field3() const;                     // ...
  const string& field4() const;
  const string& field5() const;
  ...

};
```

Now consider the cost of restoring a `LargeObject` from disk:

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);                    // restore object

  ...

}
```

Because `LargeObject` instances are big, getting all the data for such an object might be a costly database operation, especially if the data must be retrieved from a remote database and pushed across a network. In some cases, the cost of reading all that data would be unnecessary. For example, consider this kind of application:

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);
```

```
  if (object.field2() == 0) {
    cout << "Object " << id << ": null field2.\n";
  }
}
```

Here only the value of `field2` is required, so any effort spent setting up the other fields is wasted.

The lazy approach to this problem is to read no data from disk when a `LargeObject` object is created. Instead, only the "shell" of an object is created, and data is retrieved from the database only when that particular data is needed inside the object. Here's one way to implement this kind of "demand-paged" object initialization:

```
class LargeObject {
public:
  LargeObject(ObjectID id);

  const string& field1() const;
  int field2() const;
  double field3() const;
  const string& field4() const;
  ...

private:
  ObjectID oid;

  mutable string *field1Value;              // see below for a
  mutable int *field2Value;                 // discussion of "mutable"
  mutable double *field3Value;
  mutable string *field4Value;
  ...

};

LargeObject::LargeObject(ObjectID id)
: oid(id), field1Value(0), field2Value(0), field3Value(0), ...
{}

const string& LargeObject::field1() const
{
  if (field1Value == 0) {
    read the data for field 1 from the database and make
    field1Value point to it;
  }

  return *field1Value;
}
```

Each field in the object is represented as a pointer to the necessary data, and the `LargeObject` constructor initializes each pointer to null. Such null pointers signify fields that have not yet been read from the database. Each `LargeObject` member function must check the state of a field's pointer before accessing the data it points to. If the pointer is null, the corresponding data must be read from the database before performing any operations on that data.

When implementing lazy fetching, you must confront the problem that null pointers may need to be initialized to point to real data from inside any member function, including `const` member functions like `field1` . However, compilers get cranky when you try to modify data members inside `const` member functions, so you've got to find a way to say, "It's okay, I know what I'm doing." The best way to say that is to declare the pointer fields `mutable` , which means they can be modified inside any member function, even inside `const` member functions (see Item E21). That's why the fields inside `LargeObject` above are declared `mutable` .

The `mutable` keyword is a relatively recent addition to C++, so it's possible your vendors don't yet support it. If not, you'll need to find another way to convince your compilers to let you modify data members inside `const` member functions. One workable strategy is the "fake `this` " approach, whereby you create a pointer-to-non-`const` that points to the same object as `this` does. When you want to modify a data member, you access it through the "fake `this` " pointer:

```
class LargeObject {
public:
  const string& field1() const;              // unchanged
  ...

private:
  string *field1Value;                       // not declared mutable
  ...                                        // so that older
};                                           // compilers will accept it

const string& LargeObject::field1() const
{
  // declare a pointer, fakeThis, that points where this
  // does, but where the constness of the object has been
  // cast away
  LargeObject * const fakeThis =
    const_cast<LargeObject* const>(this);

  if (field1Value == 0) {
    fakeThis->field1Value =                  // this assignment is OK,
      the appropriate data                   // because what fakeThis
      from the database;                     // points to isn't const
  }

  return *field1Value;
}
```

This function employs a `const_cast` (see Item 2) to cast away the `const` ness of `*this` . If your compilers don't support `const_cast` , you can use an old C-style cast:

```
// Use of old-style cast to help emulate mutable
const string& LargeObject::field1() const
{
  LargeObject * const fakeThis = (LargeObject* const)this;

  ...                                        // as above
```

```
}
```

Look again at the pointers inside `LargeObject` . Let's face it, it's tedious and error-prone to have to initialize all those *pointers* to null, then test each one before use. Fortunately, such drudgery can be automated through the use of *smart* pointers, which you can read about in Item 28 . If you use smart pointers inside `LargeObject` , you'll also find you no longer need to declare the pointers `mutable` . Alas, it's only a temporary respite, because you'll wind up needing `mutable` once you sit down to implement the smart pointer classes. Think of it as conservation of inconvenience.

Lazy Expression Evaluation

A final example of lazy evaluation comes from numerical applications. Consider this code:

```
template<class T>
class Matrix { ... };                          // for homogeneous matrices

Matrix<int> m1(1000, 1000);                    // a 1000 by 1000 matrix
Matrix<int> m2(1000, 1000);                    // ditto

...

Matrix<int> m3 = m1 + m2;                      // add m1 and m2
```

The usual implementation of `operator+` would use eager evaluation; in this case it would compute and return the sum of `m1` and `m2` . That's a fair amount of computation (1,000,000 additions), and of course there's the cost of allocating the memory to hold all those values, too.

The lazy evaluation strategy says that's *way* too much work, so it doesn't do it. Instead, it sets up a data structure inside `m3` that indicates that `m3` 's value is the sum of `m1` and `m2` . Such a data structure might consist of nothing more than a pointer to each of `m1` and `m2` , plus an enum indicating that the operation on them is addition. Clearly, it's going to be faster to set up this data structure than to add `m1` and `m2` , and it's going to use a lot less memory, too.

Suppose that later in the program, before `m3` has been used, this code is executed:

```
Matrix<int> m4(1000, 1000);

...                                            // give m4 some values

m3 = m4 * m1;
```

Now we can forget all about `m3` being the sum of `m1` and `m2` (and thereby save the cost of the computation), and in its place we can start remembering that `m3` is the product of `m4` and `m1` . Needless to say, we don't perform the multiplication. Why bother? We're lazy, remember?

This example looks contrived, because no good programmer would write a program that computed the sum of two matrices and failed to use it, but it's not as contrived as it seems. No good programmer would deliberately compute a value that's not needed, but during maintenance, it's not

uncommon for a programmer to modify the paths through a program in such a way that a formerly useful computation becomes unnecessary. The likelihood of that happening is reduced by defining objects immediately prior to use (see Item E32), but it's still a problem that occurs from time to time.

Nevertheless, if that were the only time lazy evaluation paid off, it would hardly be worth the trouble. A more common scenario is that we need only *part* of a computation. For example, suppose we use `m3` as follows after initializing it to the sum of `m1` and `m2` :

```
cout << m3[4];                                  // print the 4th row of m3
```

Clearly we can be completely lazy no longer — we've got to compute the values in the fourth row of `m3` . But let's not be overly ambitious, either. There's no reason we have to compute any *more* than the fourth row of `m3` ; the remainder of `m3` can remain uncomputed until it's actually needed. With luck, it never will be.

How likely are we to be lucky? Experience in the domain of matrix computations suggests the odds are in our favor. In fact, lazy evaluation lies behind the wonder that is APL. APL was developed in the 1960s for interactive use by people who needed to perform matrix-based calculations. Running on computers that had less computational horsepower than the chips now found in high-end microwave ovens, APL was seemingly able to add, multiply, and even divide large matrices instantly! Its trick was lazy evaluation. The trick was usually effective, because APL users typically added, multiplied, or divided matrices not because they needed the entire resulting matrix, but only because they needed a small part of it. APL employed lazy evaluation to defer its computations until it knew exactly what part of a result matrix was needed, then it computed only that part. In practice, this allowed users to perform computationally intensive tasks *interactively* in an environment where the underlying machine was hopelessly inadequate for an implementation employing eager evaluation. Machines are faster today, but data sets are bigger and users less patient, so many contemporary matrix libraries continue to take advantage of lazy evaluation.

To be fair, laziness sometimes fails to pay off. If `m3` is used in this way,

```
cout << m3;                                     // print out all of m3
```

the jig is up and we've got to compute a complete value for `m3` . Similarly, if one of the matrices on which `m3` is dependent is about to be modified, we have to take immediate action:

```
m3 = m1 + m2;                                   // remember that m3 is the
                                                // sum of m1 and m2

m1 = m4;                                        // now m3 is the sum of m2
                                                // and the OLD value of m1!
```

Here we've got to do something to ensure that the assignment to `m1` doesn't change `m3` . Inside the

`Matrix<int>` assignment operator, we might compute `m3` 's value prior to changing `m1` or we might make a copy of the old value of `m1` and make `m3` dependent on that, but we have to do *something* to guarantee that `m3` has the value it's supposed to have after `m1` has been the target of an assignment. Other functions that might modify a matrix must be handled in a similar fashion.

Because of the need to store dependencies between values; to maintain data structures that can store values, dependencies, or a combination of the two; and to overload operators like assignment, copying, and addition, lazy evaluation in a numerical domain is a lot of work. On the other hand, it often ends up saving significant amounts of time and space during program runs, and in many applications, that's a payoff that easily justifies the significant effort lazy evaluation requires.

Summary

These four examples show that lazy evaluation can be useful in a variety of domains: to avoid unnecessary copying of objects, to distinguish reads from writes using `operator[]` , to avoid unnecessary reads from databases, and to avoid unnecessary numerical computations. Nevertheless, it's not always a good idea. Just as procrastinating on your clean-up chores won't save you any work if your parents always check up on you, lazy evaluation won't save your program any work if all your computations are necessary. Indeed, if all your computations are essential, lazy evaluation may slow you down and increase your use of memory, because, in addition to having to do all the computations you were hoping to avoid, you'll also have to manipulate the fancy data structures needed to make lazy evaluation possible in the first place. Lazy evaluation is only useful when there's a reasonable chance your software will be asked to perform computations that can be avoided.

There's nothing about lazy evaluation that's specific to C++. The technique can be applied in any programming language, and several languages — notably APL, some dialects of Lisp, and virtually all dataflow languages — embrace the idea as a fundamental part of the language. Mainstream programming languages employ eager evaluation, however, and C++ is mainstream. Yet C++ is particularly suitable as a vehicle for user-implemented lazy evaluation, because its support for encapsulation makes it possible to add lazy evaluation to a class without clients of that class knowing it's been done.

Look again at the code fragments used in the above examples, and you can verify that the class interfaces offer no hints about whether eager or lazy evaluation is used by the classes. That means it's possible to implement a class using a straightforward eager evaluation strategy, but then, if your profiling investigations (see Item 16) show that class's implementation is a performance bottleneck, you can replace its implementation with one based on lazy evaluation. (See also Item E34 .) The only change your clients will see (after recompilation or relinking) is improved performance. That's the kind of software enhancement clients love, one that can make you downright proud to be lazy.

Back to Item 16: Remember the 80-20 rule
Continue to Item 18: Amortize the cost of expected computations

Back to Item 17: Consider using lazy evaluation
Continue to Item 19: Understand the origin of temporary objects

Item 18: Amortize the cost of expected computations.

In Item 17, I extolled the virtues of laziness, of putting things off as long as possible, and I explained how laziness can improve the efficiency of your programs. In this item, I adopt a different stance. Here, laziness has no place. I now encourage you to improve the performance of your software by having it do *more* than it's asked to do. The philosophy of this item might be called *over-eager evaluation*: doing things *before* you're asked to do them.

Consider, for example, a template for classes representing large collections of numeric data:

```
template<class NumericalType>
class DataCollection {
public:
  NumericalType min() const;
  NumericalType max() const;
  NumericalType avg() const;
  ...
};
```

Assuming the `min`, `max`, and `avg` functions return the current minimum, maximum, and average values of the collection, there are three ways in which these functions can be implemented. Using eager evaluation, we'd examine all the data in the collection when `min`, `max`, or `avg` was called, and we'd return the appropriate value. Using lazy evaluation, we'd have the functions return data structures that could be used to determine the appropriate value whenever the functions' return values were actually used. Using over-eager evaluation, we'd keep track of the running minimum, maximum, and average values of the collection, so when `min`, `max`, or `avg` was called, we'd be able to return the correct value immediately — no computation would be required. If `min`, `max`, and `avg` were called frequently, we'd be able to amortize the cost of keeping track of the collection's minimum, maximum, and average values over all the calls to those functions, and the amortized cost per call would be lower than with eager or lazy evaluation.

The idea behind over-eager evaluation is that if you expect a computation to be requested frequently, you can lower the average cost per request by designing your data structures to handle the requests especially efficiently.

One of the simplest ways to do this is by caching values that have already been computed and are likely to be needed again. For example, suppose you're writing a program to provide information about employees, and one of the pieces of information you expect to be requested frequently is an employee's cubicle number. Further suppose that employee information is stored in a database, but, for most applications, an employee's cubicle number is irrelevant, so the database is not optimized to find it. To avoid having your specialized application unduly stress the database with repeated

lookups of employee cubicle numbers, you could write a `findCubicleNumber` function that caches the cubicle numbers it looks up. Subsequent requests for cubicle numbers that have already been retrieved can then be satisfied by consulting the cache instead of querying the database.

Here's one way to implement `findCubicleNumber`; it uses a `map` object from the Standard Template Library (the "STL" — see Item 35) as a local cache:

```
int findCubicleNumber(const string& employeeName)
{
  // define a static map to hold (employee name, cubicle number)
  // pairs. This map is the local cache.
  typedef map<string, int> CubicleMap;
  static CubicleMap cubes;

  // try to find an entry for employeeName in the cache;
  // the STL iterator "it" will then point to the found
  // entry, if there is one (see Item 35 for details)
  CubicleMap::iterator it = cubes.find(employeeName);

  // "it"'s value will be cubes.end() if no entry was
  // found (this is standard STL behavior). If this is
  // the case, consult the database for the cubicle
  // number, then add it to the cache
  if (it == cubes.end()) {
    int cubicle =
      the result of looking up employeeName's cubicle
      number in the database;

    cubes[employeeName] = cubicle;          // add the pair
                                            // (employeeName, cubicle)
                                            // to the cache
    return cubicle;
  }
  else {
    // "it" points to the correct cache entry, which is a
    // (employee name, cubicle number) pair. We want only
    // the second component of this pair, and the member
    // "second" will give it to us
    return (*it).second;
  }
}
```

Try not to get bogged down in the details of the STL code (which will be clearer after you've read Item 35). Instead, focus on the general strategy embodied by this function. That strategy is to use a local cache to replace comparatively expensive database queries with comparatively inexpensive lookups in an in-memory data structure. Provided we're correct in assuming that cubicle numbers will frequently be requested more than once, the use of a cache in `findCubicleNumber` should reduce the average cost of returning an employee's cubicle number.

One detail of the code requires explanation. The final statement returns `(*it).second` instead of

the more conventional `it->second`. Why? The answer has to do with the conventions followed by the STL. In brief, the iterator `it` is an object, not a pointer, so there is no guarantee that "`->`" can be applied to `it`.[6] The STL does require that "`.`" and "`*`" be valid for iterators, however, so `(*it).second`, though syntactically clumsy, is guaranteed to work.)

Caching is one way to amortize the cost of anticipated computations. Prefetching is another. You can think of prefetching as the computational equivalent of a discount for buying in bulk. Disk controllers, for example, read entire blocks or sectors of data when they read from disk, even if a program asks for only a small amount of data. That's because it's faster to read a big chunk once than to read two or three small chunks at different times. Furthermore, experience has shown that if data in one place is requested, it's quite common to want nearby data, too. This is the infamous *locality of reference* phenomenon, and systems designers rely on it to justify disk caches, memory caches for both instructions and data, and instruction prefetches.

Excuse me? You say you don't worry about such low-level things as disk controllers or CPU caches? No problem. Prefetching can yield dividends for even one as high-level as you. Imagine, for example, you'd like to implement a template for dynamic arrays, i.e., arrays that start with a size of one and automatically extend themselves so that all nonnegative indices are valid:

```
template<class T>                          // template for dynamic
class DynArray { ... };                    // array-of-T classes

DynArray<double> a;                        // at this point, only a[0]
                                           // is a legitimate array
                                           // element

a[22] = 3.5;                               // a is automatically
                                           // extended: valid indices
                                           // are now 0-22

a[32] = 0;                                 // a extends itself again;
                                           // now a[0]-a[32] are valid
```

How does a `DynArray` object go about extending itself when it needs to? A straightforward strategy would be to allocate only as much additional memory as needed, something like this:

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) {
    throw an exception;                    // negative indices are
  }                                        // still invalid

  if (index > the current maximum index value) {
    call new to allocate enough additional memory so that
```

```
    index is valid;
  }

  return the indexth element of the array;
}
```

This approach simply calls `new` each time it needs to increase the size of the array, but calls to `new` invoke `operator new` (see Item 8), and calls to `operator new` (and `operator delete`) are usually expensive. That's because they typically result in calls to the underlying operating system, and system calls are generally slower than are in-process function calls. As a result, we'd like to make as few system calls as possible.

An over-eager evaluation strategy employs this reasoning: if we have to increase the size of the array now to accommodate index *i*, the locality of reference principle suggests we'll probably have to increase it in the future to accommodate some other index a bit larger than *i*. To avoid the cost of the memory allocation for the second (anticipated) expansion, we'll increase the size of the `DynArray` now by *more* than is required to make *i* valid, and we'll hope that future expansions occur within the range we have thereby provided for. For example, we could write `DynArray::operator[]` like this:

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) throw an exception;

  if (index > the current maximum index value) {
    int diff = index - the current maximum index value;

    call new to allocate enough additional memory so that
    index+diff is valid;
  }

  return the indexth element of the array;
}
```

This function allocates twice as much memory as needed each time the array must be extended. If we look again at the usage scenario we saw earlier, we note that the `DynArray` must allocate additional memory only once, even though its logical size is extended twice:

```
DynArray<double> a;                    // only a[0] is valid

a[22] = 3.5;                           // new is called to expand
                                       // a's storage through
                                       // index 44; a's logical
                                       // size becomes 23
```

```
a[32] = 0;                                  // a's logical size is
                                            // changed to allow a[32],
                                            // but new isn't called
```

If a needs to be extended again, that extension, too, will be inexpensive, provided the new maximum index is no greater than 44.

There is a common theme running through this Item, and that's that greater speed can often be purchased at a cost of increased memory usage. Keeping track of running minima, maxima, and averages requires extra space, but it saves time. Caching results necessitates greater memory usage but reduces the time needed to regenerate the results once they've been cached. Prefetching demands a place to put the things that are prefetched, but it reduces the time needed to access those things. The story is as old as Computer Science: you can often trade space for time. (Not always, however. Using larger objects means fewer fit on a virtual memory or cache page. In rare cases, making objects bigger *reduces* the performance of your software, because your paging activity increases, your cache hit rate decreases, or both. How do you find out if you're suffering from such problems? You profile, profile, profile (see Item 16).)

The advice I proffer in this Item — that you amortize the cost of anticipated computations through over-eager strategies like caching and prefetching — is not contradictory to the advice on lazy evaluation I put forth in Item 17. Lazy evaluation is a technique for improving the efficiency of programs when you must support operations whose results are *not always* needed. Over-eager evaluation is a technique for improving the efficiency of programs when you must support operations whose results are *almost always* needed or whose results are often needed more than once. Both are more difficult to implement than run-of-the-mill eager evaluation, but both can yield significant performance improvements in programs whose behavioral characteristics justify the extra programming effort.

Back to Item 17: Consider using lazy evaluation
Continue to Item 19: Understand the origin of temporary objects

---

[6] In July 1995, the ◦ISO/ANSI committee standardizing C++ added a requirement that STL iterators support the "`->`" operator, so `it->second` should now work. Some STL implementations fail to satisfy this requirement, however, so `(*it).second` is still the more portable construct.
Return

Back to Item 18: Amortize the cost of expected computations
Continue to Item 20: Facilitate the return value optimization

Item 19: Understand the origin of temporary objects.

When programmers speak amongst themselves, they often refer to variables that are needed for only a short while as "temporaries." For example, in this `swap` routine,

```
template<class T>
void swap(T& object1, T& object2)
{
  T temp = object1;
  object1 = object2;
  object2 = temp;
}
```

it's common to call `temp` a "temporary." As far as C++ is concerned, however, `temp` is not a temporary at all. It's simply an object local to a function.

True temporary objects in C++ are invisible — they don't appear in your source code. They arise whenever a non-heap object is created but not named. Such *unnamed* objects usually arise in one of two situations: when implicit type conversions are applied to make function calls succeed and when functions return objects. It's important to understand how and why these temporary objects are created and destroyed, because the attendant costs of their construction and destruction can have a noticeable impact on the performance of your programs.

Consider first the case in which temporary objects are created to make function calls succeed. This happens when the type of object passed to a function is not the same as the type of the parameter to which it is being bound. For example, consider a function that counts the number of occurrences of a character in a string:

```
// returns the number of occurrences of ch in str
size_t countChar(const string& str, char ch);

char buffer[MAX_STRING_LEN];
char c;

// read in a char and a string; use setw to avoid
// overflowing buffer when reading the string
cin >> c >> setw(MAX_STRING_LEN) >> buffer;

cout << "There are " << countChar(buffer, c)
     << " occurrences of the character " << c
     << " in " << buffer << endl;
```

Look at the call to `countChar`. The first argument passed is a `char` array, but the corresponding function parameter is of type `const string&`. This call can succeed only if the type mismatch can be eliminated, and your compilers will be happy to eliminate it by creating a temporary object of type `string`. That temporary object is initialized by calling the `string` constructor with `buffer` as its argument. The `str` parameter of `countChar` is then bound to this temporary `string` object. When `countChar` returns, the temporary object is automatically destroyed.

Conversions such as these are convenient (though dangerous — see Item 5), but from an efficiency point of view, the construction and destruction of a temporary `string` object is an unnecessary expense. There are two general ways to eliminate it. One is to redesign your code so conversions like these can't take place. That strategy is examined in Item 5. An alternative tack is to modify your software so that the conversions are unnecessary. Item 21 describes how you can do that.

These conversions occur only when passing objects by value or when passing to a reference-to-`const` parameter. They do not occur when passing an object to a reference-to-non-`const` parameter. Consider this function:

```
void uppercasify(string& str);              // changes all chars in
                                            // str to upper case
```

In the character-counting example, a `char` array could be successfully passed to `countChar`, but here, trying to call `uppercasify` with a `char` array fails:

```
char subtleBookPlug[] = "Effective C++";

uppercasify(subtleBookPlug);                // error!
```

No temporary is created to make the call succeed. Why not?

Suppose a temporary were created. Then the temporary would be passed to `uppercasify`, which would modify the temporary so its characters were in upper case. But the actual argument to the function call — `subtleBookPlug` — would *not be affected*; only the temporary `string` object generated from `subtleBookPlug` would be changed. Surely this is not what the programmer intended. That programmer passed `subtleBookPlug` to `uppercasify`, and that programmer expected `subtleBookPlug` to be modified. Implicit type conversion for references-to-non-`const` objects, then, would allow temporary objects to be changed when programmers expected non-temporary objects to be modified. That's why the language prohibits the generation of temporaries for non-`const` reference parameters. Reference-to-`const` parameters don't suffer from this problem, because such parameters, by virtue of being `const`, can't be changed.

The second set of circumstances under which temporary objects are created is when a function returns an object. For instance, `operator+` must return an object that represents the sum of its operands (see Item E23). Given a type `Number`, for example, `operator+` for that type would be declared like this:

```
const Number operator+(const Number& lhs,
                       const Number& rhs);
```

The return value of this function is a temporary, because it has no name: it's just the function's return value. You must pay to construct and destruct this object each time you call `operator+`. (For an explanation of why the return value is `const`, see Item E21.)

As usual, you don't want to incur this cost. For this particular function, you can avoid paying by switching to a similar function, `operator+=`; Item 22 tells you about this transformation. For most functions that return objects, however, switching to a different function is not an option and there is no way to avoid the construction and destruction of the return value. At least, there's no way to avoid it *conceptually*. Between concept and reality, however, lies a murky zone called *optimization*, and sometimes you can write your object-returning functions in a way that allows your compilers to optimize temporary objects out of existence. Of these optimizations, the most common and useful is the *return value optimization*, which is the subject of Item 20.

The bottom line is that temporary objects can be costly, so you want to eliminate them whenever you can. More important than this, however, is to train yourself to look for places where temporary objects may be created. Anytime you see a reference-to-`const` parameter, the possibility exists that a temporary will be created to bind to that parameter. Anytime you see a function returning an object, a temporary will be created (and later destroyed). Learn to look for such constructs, and your insight into the cost of "behind the scenes" compiler actions will markedly improve.

Back to Item 18: Amortize the cost of expected computations
Continue to Item 20: Facilitate the return value optimization

Back to Item 19: Understand the origin of temporary objects
Continue to Item 21: Overload to avoid implicit type conversions

Item 20: Facilitate the return value optimization.

A function that returns an object is frustrating to efficiency aficionados, because the by-value return, including the constructor and destructor calls it implies (see Item 19), cannot be eliminated. The problem is simple: a function either has to return an object in order to offer correct behavior or it doesn't. If it does, there's no way to get rid of the object being returned. Period.

Consider the `operator*` function for rational numbers:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  ...
  int numerator() const;
  int denominator() const;
};

// For an explanation of why the return value is const,
// see Item 6
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Without even looking at the code for `operator*`, we know it must return an object, because it returns the product of two arbitrary numbers. These are *arbitrary* numbers. How can `operator*` possibly avoid creating a new object to hold their product? It can't, so it must create a new object and return it. C++ programmers have nevertheless expended Herculean efforts in a search for the legendary elimination of the by-value return (see Items E23 and E31).

Sometimes people return pointers, which leads to this syntactic travesty:

```
// an unreasonable way to avoid returning an object
const Rational * operator*(const Rational& lhs,
                           const Rational& rhs);

Rational a = 10;
Rational b(1, 2);

Rational c = *(a * b);                      // Does this look "natural"
                                            // to you?
```

It also raises a question. Should the caller delete the pointer returned by the function? The answer is

usually yes, and that usually leads to resource leaks.

Other developers return references. That yields an acceptable syntax,

```
// a dangerous (and incorrect) way to avoid returning
// an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs);

Rational a = 10;
Rational b(1, 2);

Rational c = a * b;                          // looks perfectly reasonable
```

but such functions can't be implemented in a way that behaves correctly. A common attempt looks like this:

```
// another dangerous (and incorrect) way to avoid
// returning an object
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs)
{
  Rational result(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
  return result;
}
```

This function returns a reference to an object that no longer exists. In particular, it returns a reference to the local object `result`, but `result` is automatically destroyed when `operator*` is exited. Returning a reference to an object that's been destroyed is hardly useful.

Trust me on this: some functions (`operator*` among them) just have to return objects. That's the way it is. Don't fight it. You can't win.

That is, you can't win in your effort to eliminate by-value returns from functions that require them. But that's the wrong war to wage. From an efficiency point of view, you shouldn't care that a function returns an object, you should only care about the *cost* of that object. What you need to do is channel your efforts into finding a way to reduce the cost of returned objects, not to eliminate the objects themselves (which we now recognize is a futile quest). If no cost is associated with such objects, who cares how many get created?

It is frequently possible to write functions that return objects in such a way that compilers can eliminate the cost of the temporaries. The trick is to return *constructor arguments* instead of

objects, and you can do it like this:

```
// an efficient and correct way to implement a
// function that returns an object
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

Look closely at the expression being returned. It looks like you're calling a `Rational` constructor, and in fact you are. You're creating a temporary `Rational` object through this expression,

```
Rational(lhs.numerator() * rhs.numerator(),
         lhs.denominator() * rhs.denominator());
```

and it is this temporary object the function is copying for its return value.

This business of returning constructor arguments instead of local objects doesn't appear to have bought you a lot, because you still have to pay for the construction and destruction of the temporary created inside the function, and you still have to pay for the construction and destruction of the object the function returns. But you have gained something. The rules for C++ allow compilers to optimize temporary objects out of existence. As a result, if you call `operator*` in a context like this,

```
Rational a = 10;
Rational b(1, 2);

Rational c = a * b;                          // operator* is called here
```

your compilers are allowed to eliminate both the temporary inside `operator*` *and* the temporary returned by `operator*`. They can construct the object defined by the `return` expression *inside the memory allotted for the object `c`*. If your compilers do this, the total cost of temporary objects as a result of your calling `operator*` is zero: no temporaries are created. Instead, you pay for only one constructor call — the one to create `c`. Furthermore, you can't do any better than this, because `c` is a named object, and named objects can't be eliminated (see also Item 22).[7] You can, however, eliminate the overhead of the call to `operator*` by declaring that function `inline` (but first see Item E33):

```
// the most efficient way to write a function returning
// an object
```

```
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers *can* do? I want to know what they *do* do. Does any of this nonsense work with real compilers?" It does. This particular optimization — eliminating a local temporary by using a function's return location (and possibly replacing that with an object at the function's call site) — is both well-known and commonly implemented. It even has a name: the *return value optimization*. In fact, the existence of a name for this optimization may explain why it's so widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it.

Back to Item 19: Understand the origin of temporary objects
Continue to Item 21: Overload to avoid implicit type conversions

---

[7] In July 1996, the ISO/ANSI standardization committee declared that both named and unnamed objects may be optimized away via the return value optimization, so both versions of `operator*` above may now yield the same (optimized) object code.
Return

Back to Item 20: Facilitate the return value optimization
Continue to Item 22: Consider using op= instead of stand-alone op

Item 21: Overload to avoid implicit type conversions.

Here's some code that looks nothing if not eminently reasonable:

```
class UPInt {                                   // class for unlimited
public:                                         // precision integers
  UPInt();
  UPInt(int value);
  ...

};

// For an explanation of why the return value is const,
// see Item E21
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;
```

There are no surprises here. `upi1` and `upi2` are both `UPInt` objects, so adding them together just calls `operator+` for `UPInts`.

Now consider these statements:

```
upi3 = upi1 + 10;

upi3 = 10 + upi2;
```

These statements also succeed. They do so through the creation of temporary objects to convert the integer 10 into `UPInts` (see Item 19).

It is convenient to have compilers perform these kinds of conversions, but the temporary objects created to make the conversions work are a cost we may not wish to bear. Just as most people want government benefits without having to pay for them, most C++ programmers want implicit type conversions without incurring any cost for temporaries. But without the computational equivalent of deficit spending, how can we do it?

We can take a step back and recognize that our goal isn't really type conversion, it's being able to

make calls to `operator+` with a combination of `UPInt` and `int` arguments. Implicit type conversion happens to be a means to that end, but let us not confuse means and ends. There is another way to make mixed-type calls to `operator+` succeed, and that's to eliminate the need for type conversions in the first place. If we want to be able to add `UPInt` and `int` objects, all we have to do is say so. We do it by declaring *several* functions, each with a different set of parameter types:

```
const UPInt operator+(const UPInt& lhs,      // add UPInt
                      const UPInt& rhs);     // and UPInt

const UPInt operator+(const UPInt& lhs,      // add UPInt
                      int rhs);              // and int

const UPInt operator+(int lhs,               // add int and
                      const UPInt& rhs);     // UPInt

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;                    // fine, no temporary for
                                             // upi1 or upi2

upi3 = upi1 + 10;                            // fine, no temporary for
                                             // upi1 or 10

upi3 = 10 + upi2;                            // fine, no temporary for
                                             // 10 or upi2
```

Once you start overloading to eliminate type conversions, you run the risk of getting swept up in the passion of the moment and declaring functions like this:

```
const UPInt operator+(int lhs, int rhs);          // error!
```

The thinking here is reasonable enough. For the types `UPInt` and `int`, we want to overload on all possible combinations for `operator+`. Given the three overloadings above, the only one missing is `operator+` taking two `int` arguments, so we want to add it.

Reasonable or not, there are rules to this C++ game, and one of them is that every overloaded operator must take at least one argument of a user-defined type. `int` isn't a user-defined type, so we can't overload an operator taking only arguments of that type. (If this rule didn't exist, programmers would be able to change the meaning of predefined operations, and that would surely lead to chaos. For example, the attempted overloading of `operator+` above would change the meaning of addition on `int`s. Is that really something we want people to be able to do?)

Overloading to avoid temporaries isn't limited to operator functions. For example, in most

programs, you'll want to allow a `string` object everywhere a `char*` is acceptable, and vice versa. Similarly, if you're using a numerical class like `complex` (see Item 35), you'll want types like `int` and `double` to be valid anywhere a numerical object is. As a result, any function taking arguments of type `string`, `char*`, `complex`, etc., is a reasonable candidate for overloading to eliminate type conversions.

Still, it's important to keep the 80-20 rule (see Item 16) in mind. There is no point in implementing a slew of overloaded functions unless you have good reason to believe that it will make a noticeable improvement in the overall efficiency of the programs that use them.

Back to Item 20: Facilitate the return value optimization
Continue to Item 22: Consider using op= instead of stand-alone op

Back to Item 21: Overload to avoid implicit type conversions
Continue to Item 23: Consider alternative libraries

Item 22: Consider using *op*= instead of stand-alone *op*.

Most programmers expect that if they can say things like these,

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

they can also say things like these:

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

If `x` and `y` are of a user-defined type, there is no guarantee that this is so. As far as C++ is concerned, there is no relationship between `operator+`, `operator=`, and `operator+=`, so if you want all three operators to exist and to have the expected relationship, you must implement that yourself. Ditto for the operators `-`, `*`, `/`, etc.

A good way to ensure that the natural relationship between the assignment version of an operator (e.g., `operator+=`) and the stand-alone version (e.g., `operator+`) exists is to implement the latter in terms of the former (see also Item 6). This is easy to do:

```
class Rational {
public:
  ...
  Rational& operator+=(const Rational& rhs);
  Rational& operator-=(const Rational& rhs);
};

// operator+ implemented in terms of operator+=; see
// Item E21 for an explanation of why the return value is
// const and page 109 for a warning about implementation
const Rational operator+(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) += rhs;
}

// operator- implemented in terms of operator -=
const Rational operator-(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) -= rhs;
}
```

In this example, operators `+=` and `-=` are implemented (elsewhere) from scratch, and `operator+` and `operator-` call them to provide their own functionality. With this design, only the assignment versions of these operators need to be maintained. Furthermore, assuming the assignment versions of the operators are in the class's public interface, there is never a need for the stand-alone operators to be friends of the class (see Item E19).

If you don't mind putting all stand-alone operators at global scope, you can use templates to eliminate the need to write the stand-alone functions:

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  return T(lhs) += rhs;                    // see discussion below
}

template<class T>
const T operator-(const T& lhs, const T& rhs)
{
  return T(lhs) -= rhs;                     // see discussion below
}

...
```

With these templates, as long as an assignment version of an operator is defined for some type T, the corresponding stand-alone operator will automatically be generated if it's needed.

All this is well and good, but so far we have failed to consider the issue of efficiency, and efficiency is, after all, the topic of this chapter. Three aspects of efficiency are worth noting here. The first is that, in general, assignment versions of operators are more efficient than stand-alone versions, because stand-alone versions must typically return a new object, and that costs us the construction and destruction of a temporary (see Items 19 and 20, as well as Item E23). Assignment versions of operators write to their left-hand argument, so there is no need to generate a temporary to hold the operator's return value.

The second point is that by offering assignment versions of operators as well as stand-alone versions, you allow *clients* of your classes to make the difficult trade-off between efficiency and convenience. That is, your clients can decide whether to write their code like this,

```
Rational a, b, c, d, result;
...
result = a + b + c + d;                     // probably uses 3 temporary
                                            // objects, one for each call
                                            // to operator+
```

or like this:

```
result = a;                                   // no temporary needed
result += b;                                  // no temporary needed
result += c;                                  // no temporary needed
result += d;                                  // no temporary needed
```

The former is easier to write, debug, and maintain, and it offers acceptable performance about 80% of the time (see Item 16). The latter is more efficient, and, one supposes, more intuitive for assembly language programmers. By offering both options, you let clients develop and debug code using the easier-to-read stand-alone operators while still reserving the right to replace them with the more efficient assignment versions of the operators. Furthermore, by implementing the stand-alones in terms of the assignment versions, you ensure that when clients switch from one to the other, the semantics of the operations remain constant.

The final efficiency observation concerns implementing the stand-alone operators. Look again at the implementation for `operator+`:

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{ return T(lhs) += rhs; }
```

The expression `T(lhs)` is a call to `T`'s copy constructor. It creates a temporary object whose value is the same as that of `lhs`. This temporary is then used to invoke `operator+=` with `rhs`, and the result of that operation is returned from `operator+`.[8] This code seems unnecessarily cryptic. Wouldn't it be better to write it like this?

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  T result(lhs);                              // copy lhs into result
  return result += rhs;                       // add rhs to it and return
}
```

This template is almost equivalent to the one above, but there is a crucial difference. This second template contains a named object, `result`. The fact that this object is named means that the return value optimization (see Item 20) was, until relatively recently, unavailable for this implementation of `operator+` (see the footnote on page 104). The first implementation has *always* been eligible for the return value optimization, so the odds may be better that the compilers you use will generate optimized code for it.

Now, truth in advertising compels me to point out that the expression

```
return T(lhs) += rhs;
```

is more complex than most compilers are willing to subject to the return value optimization. The first implementation above may thus cost you one temporary object within the function, just as you'd pay for using the named object `result`. However, the fact remains that unnamed objects have historically been easier to eliminate than named objects, so when faced with a choice between a named object and a temporary object, you may be better off using the temporary. It should never cost you more than its named colleague, and, especially with older compilers, it may cost you less.

All this talk of named objects, unnamed objects, and compiler optimizations is interesting, but let us not forget the big picture. The big picture is that assignment versions of operators (such as `operator+=`) tend to be more efficient than stand-alone versions of those operators (e.g. `operator+`). As a library designer, you should offer both, and as an application developer, you should consider using assignment versions of operators instead of stand-alone versions whenever performance is at a premium.

Back to Item 21: Overload to avoid implicit type conversions
Continue to Item 23: Consider alternative libraries

---

[8] At least that's what's supposed to happen. Alas, some compilers treat `T(lhs)` as a *cast* to remove `lhs`'s `const`ness, then add `rhs` to `lhs` and return a reference to the modified `lhs`! Test your compilers before relying on the behavior described above.
Return

Back to Item 22: Consider using op= instead of stand-alone op
Continue to Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Item 23: Consider alternative libraries.

Library design is an exercise in compromise. The ideal library is small, fast, powerful, flexible, extensible, intuitive, universally available, well supported, free of use restrictions, and bug-free. It is also nonexistent. Libraries optimized for size and speed are typically not portable. Libraries with rich functionality are rarely intuitive. Bug-free libraries are limited in scope. In the real world, you can't have everything; something always has to give.

Different designers assign different priorities to these criteria. They thus sacrifice different things in their designs. As a result, it is not uncommon for two libraries offering similar functionality to have quite different performance profiles.

As an example, consider the iostream and stdio libraries, both of which should be available to every C++ programmer. The iostream library has several advantages over its C counterpart (see Item E2). It's type-safe, for example, and it's extensible. In terms of efficiency, however, the iostream library generally suffers in comparison with stdio, because stdio usually results in executables that are both smaller and faster than those arising from iostreams.

Consider first the speed issue. One way to get a feel for the difference in performance between iostreams and stdio is to run benchmark applications using both libraries. Now, it's important to bear in mind that benchmarks lie. Not only is it difficult to come up with a set of inputs that correspond to "typical" usage of a program or library, it's also useless unless you have a reliable way of determining how "typical" you or your clients are. Nevertheless, benchmarks can provide *some* insight into the comparative performance of different approaches to a problem, so though it would be foolish to rely on them completely, it would also be foolish to ignore them.

Let's examine a simple-minded benchmark program that exercises only the most rudimentary I/O functionality. This program reads 30,000 floating point numbers from standard input and writes them to standard output in a fixed format. The choice between the iostream and stdio libraries is made during compilation and is determined by the preprocessor symbol `STDIO`. If this symbol is defined, the stdio library is used, otherwise the iostream library is employed.

```
#ifdef STDIO
#include <stdio.h>
#else
#include <iostream>
#include <iomanip>
using namespace std;
#endif
```

```
const int VALUES = 30000;                // # of values to read/write

int main()
{
  double d;

  for (int n = 1; n <= VALUES; ++n) {
#ifdef STDIO
    scanf("%lf", &d);
    printf("%10.5f", d);
#else
    cin >> d;
    cout  << setw(10)                    // set field width
          << setprecision(5)             // set decimal places
          << setiosflags(ios::showpoint) // keep trailing 0s
          << setiosflags(ios::fixed)     // use these settings
          << d;
#endif

    if (n % 5 == 0) {
#ifdef STDIO
      printf("\n");
#else
      cout << '\n';
#endif
    }
  }

  return 0;
}
```

When this program is given the natural logarithms of the positive integers as input, it produces output like this:

```
0.00000   0.69315   1.09861   1.38629   1.60944
1.79176   1.94591   2.07944   2.19722   2.30259
2.39790   2.48491   2.56495   2.63906   2.70805
2.77259   2.83321   2.89037   2.94444   2.99573
3.04452   3.09104   3.13549   3.17805   3.21888
```

Such output demonstrates, if nothing else, that it's possible to produce fixed-format I/O using iostreams. Of course,

```
cout  << setw(10)
      << setprecision(5)
      << setiosflags(ios::showpoint)
      << setiosflags(ios::fixed)
      << d;
```

is nowhere near as easy to type as

```
printf("%10.5f", d);
```

but `operator<<` is both type-safe and extensible, and `printf` is neither.

I have run this program on several combinations of machines, operating systems, and compilers, and in every case the stdio version has been faster. Sometimes it's been only a little faster (about 20%), sometimes it's been substantially faster (nearly 200%), but I've never come across an iostream implementation that was as fast as the corresponding stdio implementation. In addition, the size of this trivial program's executable using stdio tends to be smaller (sometimes *much* smaller) than the corresponding program using iostreams. (For programs of a realistic size, this difference is rarely significant.)

Bear in mind that any efficiency advantages of stdio are highly implementation-dependent, so future implementations of systems I've tested or existing implementations of systems I haven't tested may show a negligible performance difference between iostreams and stdio. In fact, one can reasonably hope to discover an iostream implementation that's *faster* than stdio, because iostreams determine the types of their operands during compilation, while stdio functions typically parse a format string at runtime.

The contrast in performance between iostreams and stdio is just an example, however, it's not the main point. The main point is that different libraries offering similar functionality often feature different performance trade-offs, so once you've identified the bottlenecks in your software (via profiling — see Item 16), you should see if it's possible to remove those bottlenecks by replacing one library with another. If your program has an I/O bottleneck, for example, you might consider replacing iostreams with stdio, but if it spends a significant portion of its time on dynamic memory allocation and deallocation, you might see if there are alternative implementations of `operator new` and `operator delete` available (see Item 8 and Item E10). Because different libraries embody different design decisions regarding efficiency, extensibility, portability, type safety, and other issues, you can sometimes significantly improve the efficiency of your software by switching to libraries whose designers gave more weight to performance considerations than to other factors.

Back to Item 22: Consider using op= instead of stand-alone op
Continue to Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Back to Item 23: Consider alternative libraries
Continue to Techniques

Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.

C++ compilers must find a way to implement each feature in the language. Such implementation details are, of course, compiler-dependent, and different compilers implement language features in different ways. For the most part, you need not concern yourself with such matters. However, the implementation of some features can have a noticeable impact on the size of objects and the speed at which member functions execute, so for those features, it's important to have a basic understanding of what compilers are likely to be doing under the hood. The foremost example of such a feature is virtual functions.

When a virtual function is called, the code executed must correspond to the dynamic type of the object on which the function is invoked; the type of the pointer or reference to the object is immaterial. How can compilers provide this behavior efficiently? Most implementations use *virtual tables* and *virtual table pointers*. Virtual tables and virtual table pointers are commonly referred to as *vtbls* and *vptrs*, respectively.

A vtbl is usually an array of pointers to functions. (Some compilers use a form of linked list instead of an array, but the fundamental strategy is the same.) Each class in a program that declares or inherits virtual functions has its own vtbl, and the entries in a class's vtbl are pointers to the implementations of the virtual functions for that class. For example, given a class definition like this,

```
class C1 {
public:
  C1();

  virtual ~C1();
  virtual void f1();
  virtual int f2(char c) const;
  virtual void f3(const string& s);

  void f4() const;

  ...
};
```

`C1`'s virtual table array will look something like this:

Note that the nonvirtual function `f4` is not in the table, nor is `C1`'s constructor. Nonvirtual functions — including constructors, which are by definition nonvirtual — are implemented just like ordinary C functions, so there are no special performance considerations surrounding their use.

If a class `C2` inherits from `C1`, redefines some of the virtual functions it inherits, and adds some new ones of its own,

```
class C2: public C1 {
public:
  C2();                                    // nonvirtual function
  virtual ~C2();                           // redefined function
  virtual void f1();                       // redefined function
  virtual void f5(char *str);              // new virtual function
  ...
};
```

its virtual table entries point to the functions that are appropriate for objects of its type. These entries include pointers to the `C1` virtual functions that `C2` chose not to redefine:

This discussion brings out the first cost of virtual functions: you have to set aside space for a virtual table for each class that contains virtual functions. The size of a class's vtbl is proportional to the number of virtual functions declared for that class (including those it inherits from its base classes). There should be only one virtual table per class, so the total amount of space required for virtual tables is not usually significant, but if you have a large number of classes or a large number of virtual functions in each class, you may find that the vtbls take a significant bite out of your address space.

Because you need only one copy of a class's vtbl in your programs, compilers must address a tricky problem: where to put it. Most programs and libraries are created by linking together many object files, but each object file is generated independently of the others. Which object file should contain

the vtbl for any given class? You might think to put it in the object file containing `main`, but libraries have no `main`, and at any rate the source file containing `main` may make no mention of many of the classes requiring vtbls. How could compilers then know which vtbls they were supposed to create?

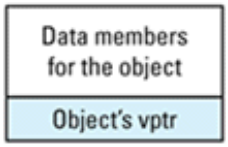
A different strategy must be adopted, and compiler vendors tend to fall into two camps. For vendors who provide an integrated environment containing both compiler and linker, a brute-force strategy is to generate a copy of the vtbl in each object file that might need it. The linker then strips out duplicate copies, leaving only a single instance of each vtbl in the final executable or library.

A more common design is to employ a heuristic to determine which object file should contain the vtbl for a class. Usually this heuristic is as follows: a class's vtbl is generated in the object file containing the definition (i.e., the body) of the first non-inline non-pure virtual function in that class. Thus, the vtbl for class `C1` above would be placed in the object file containing the definition of `C1::~C1` (provided that function wasn't `inline`), and the vtbl for class `C2` would be placed in the object file containing the definition of `C2::~C2` (again, provided that function wasn't `inline`).

In practice, this heuristic works well, but you can get into trouble if you go overboard on declaring virtual functions `inline` (see Item E33). If all virtual functions in a class are declared `inline`, the heuristic fails, and most heuristic-based implementations then generate a copy of the class's vtbl in *every object file* that uses it. In large systems, this can lead to programs containing hundreds or thousands of copies of a class's vtbl! Most compilers following this heuristic give you some way to control vtbl generation manually, but a better solution to this problem is to avoid declaring virtual functions `inline`. As we'll see below, there are good reasons why present compilers typically ignore the `inline` directive for virtual functions, anyway.

Virtual tables are half the implementation machinery for virtual functions, but by themselves they are useless. They become useful only when there is some way of indicating which vtbl corresponds to each object, and it is the job of the virtual table pointer to establish that correspondence.

Each object whose class declares virtual functions carries with it a hidden data member that points to the virtual table for that class. This hidden data member — the *vptr* — is added by compilers at a location in the object known only to the compilers. Conceptually, we can think of the layout of an object that has virtual functions as looking like this:

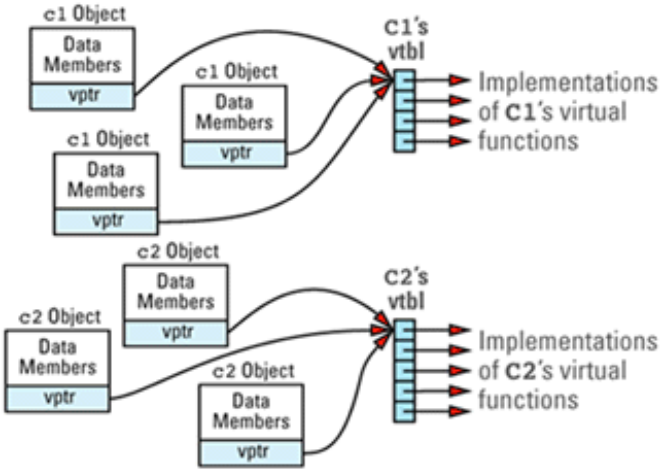


This picture shows the vptr at the end of the object, but don't be fooled: different compilers put

them in different places. In the presence of inheritance, an object's vptr is often surrounded by data members. Multiple inheritance complicates this picture, but we'll deal with that a bit later. At this point, simply note the second cost of virtual functions: you have to pay for an extra pointer inside each object that is of a class containing virtual functions.

If your objects are small, this can be a significant cost. If your objects contain, on average, four bytes of member data, for example, the addition of a vptr can *double* their size (assuming four bytes are devoted to the vptr). On systems with limited memory, this means the number of objects you can create is reduced. Even on systems with unconstrained memory, you may find that the performance of your software decreases, because larger objects mean fewer fit on each cache or virtual memory page, and that means your paging activity will probably increase.

Suppose we have a program with several objects of types `C1` and `C2`. Given the relationships among objects, vptrs, and vtbls that we have just seen, we can envision the objects in our program like this:



Now consider this program fragment:

```
void makeACall(C1 *pC1)
{
  pC1->f1();
}
```

This is a call to the virtual function `f1` through the pointer `pC1`. By looking only at this code, there is no way to know which `f1` function — `C1::f1` or `C2::f1` — should be invoked, because `pC1` might point to a `C1` object or to a `C2` object. Your compilers must nevertheless generate code for the call to `f1` inside `makeACall`, and they must ensure that the correct function is called, no matter what `pC1` points to. They do this by generating code to do the following:

1. Follow the object's vptr to its vtbl. This is a simple operation, because the compilers know where to look inside the object for the vptr. (After all, they put it there.) As a result, this costs only an offset adjustment (to get to the vptr) and a pointer indirection (to get to the vtbl).
2. Find the pointer in the vtbl that corresponds to the function being called (`f1` in this example). This, too, is simple, because compilers assign each virtual function a unique index within the table. The cost of this step is just an offset into the vtbl array.
3. Invoke the function pointed to by the pointer located in step 2.

If we imagine that each object has a hidden member called `vptr` and that the vtbl index of function `f1` is `i`, the code generated for the statement

```
pC1->f1();
```

is

```
(*pC1->vptr[i])(pC1);           // call the function pointed to by the
                                // i-th entry in the vtbl pointed to
                                // by pC1->vptr; pC1 is passed to the
                                // function as the "this" pointer
```

This is almost as efficient as a non-virtual function call: on most machines it executes only a few more instructions. The cost of calling a virtual function is thus basically the same as that of calling a function through a function pointer. Virtual functions *per se* are not usually a performance bottleneck.

The real runtime cost of virtual functions has to do with their interaction with inlining. For all practical purposes, virtual functions aren't inlined. That's because "inline" means "during compilation, replace the call site with the body of the called function," but "virtual" means "wait until runtime to see which function is called." If your compilers don't know which function will be called at a particular call site, you can understand why they won't inline that function call. This is the third cost of virtual functions: you effectively give up inlining. (Virtual functions can be inlined when invoked through *objects*, but most virtual function calls are made through *pointers* or *references* to objects, and such calls are not inlined. Because such calls are the norm, virtual functions are effectively not inlined.)

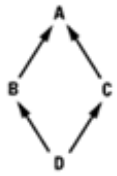
Everything we've seen so far applies to both single and multiple inheritance, but when multiple

inheritance enters the picture, things get more complex (see Item E43). There is no point in dwelling on details, but with multiple inheritance, offset calculations to find vptrs within objects become more complicated; there are multiple vptrs within a single object (one per base class); and special vtbls must be generated for base classes in addition to the stand-alone vtbls we have discussed. As a result, both the per-class and the per-object space overhead for virtual functions increases, and the runtime invocation cost grows slightly, too.

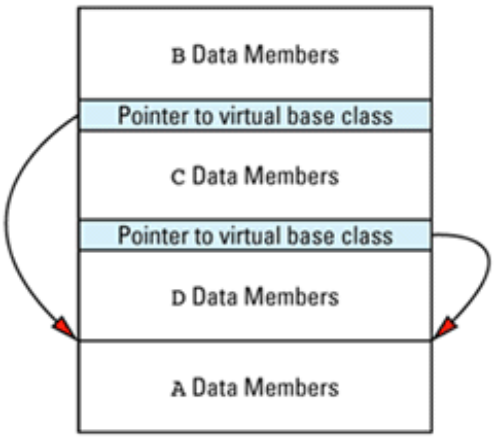
Multiple inheritance often leads to the need for virtual base classes. Without virtual base classes, if a derived class has more than one inheritance path to a base class, the data members of that base class are replicated within each derived class object, one copy for each path between the derived class and the base class. Such replication is almost never what programmers want, and making base classes virtual eliminates the replication. Virtual base classes may incur a cost of their own, however, because implementations of virtual base classes often use pointers to virtual base class parts as the means for avoiding the replication, and one or more of those pointers may be stored inside your objects.

For example, consider this, which I generally call "the dreaded multiple inheritance diamond:"

```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

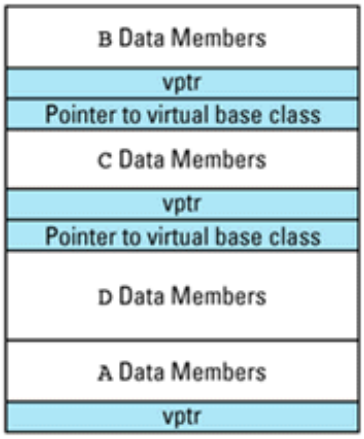


Here `A` is a virtual base class because `B` and `C` virtually inherit from it. With some compilers (especially older compilers), the layout for an object of type `D` is likely to look like this:



It seems a little strange to place the base class data members at the end of the object, but that's often how it's done. Of course, implementations are free to organize memory any way they like, so you should never rely on this picture for anything more than a conceptual overview of how virtual base classes may lead to the addition of hidden pointers to your objects. Some implementations add fewer pointers, and some find ways to add none at all. (Such implementations make the vptr and vtbl serve double duty).

If we combine this picture with the earlier one showing how virtual table pointers are added to objects, we realize that if the base class `A` in the hierarchy on page 119 has any virtual functions, the memory layout for an object of type `D` could look like this:



Here I've shaded the parts of the object that are added by compilers. The picture may be misleading, because the ratio of shaded to unshaded areas is determined by the amount of data in your classes. For small classes, the relative overhead is large. For classes with more data, the relative overhead is less significant, though it is typically noticeable.

An oddity in the above diagram is that there are only three vptrs even though four classes are involved. Implementations are free to generate four vptrs if they like, but three suffice (it turns out that `B` and `D` can share a vptr), and most implementations take advantage of this opportunity to reduce the compiler-generated overhead.

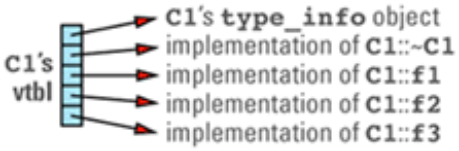
We've now seen how virtual functions make objects larger and preclude inlining, and we've examined how multiple inheritance and virtual base classes can also increase the size of objects. Let us therefore turn to our final topic, the cost of runtime type identification (RTTI).

RTTI lets us discover information about objects and classes at runtime, so there has to be a place to store the information we're allowed to query. That information is stored in an object of type `type_info`, and you can access the `type_info` object for a class by using the `typeid` operator.

There only needs to be a single copy of the RTTI information for each class, but there must be a way to get to that information for any object. Actually, that's not quite true. The language specification states that we're guaranteed accurate information on an object's dynamic type only if that type has at least one virtual function. This makes RTTI data sound a lot like a virtual function table. We need only one copy of the information per class, and we need a way to get to the appropriate information from any object containing a virtual function. This parallel between RTTI and virtual function tables is no accident: RTTI was designed to be implementable in terms of a

class's vtbl.

For example, index 0 of a vtbl array might contain a pointer to the `type_info` object for the class corresponding to that vtbl. The vtbl for class `C1` on page 114 would then look like this:



With this implementation, the space cost of RTTI is an additional entry in each class vtbl plus the cost of the storage for the `type_info` object for each class. Just as the memory for virtual tables is unlikely to be noticeable for most applications, however, you're unlikely to run into problems due to the size of `type_info` objects.

The following table summarizes the primary costs of virtual functions, multiple inheritance, virtual base classes, and RTTI:

Feature	Increases Size of Objects	Increases Per-Class Data	Reduces Inlining
Virtual Functions	Yes	Yes	Yes
Multiple Inheritance	Yes	Yes	No
Virtual Base Classes	Often	Sometimes	No
RTTI	No	Yes	No

Some people look at this table and are aghast. "I'm sticking with C!", they declare. Fair enough. But remember that each of these features offers functionality you'd otherwise have to code by hand. In most cases, your manual approximation would probably be less efficient and less robust than the compiler-generated code. Using nested `switch` statements or cascading `if-then-else`s to emulate virtual function calls, for example, yields more code than virtual function calls do, and the code runs more slowly, too. Furthermore, you must manually track object types yourself, which means your objects carry around type tags of their own; you thus often fail to gain even the benefit of smaller objects.

It is important to understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI, but it is equally important to understand that if you need the functionality these features offer, you *will* pay for it, one way or another. Sometimes you have legitimate reasons for bypassing the compiler-generated services. For example, hidden vptrs and pointers to virtual base classes can make it difficult to store C++ objects in databases or to move them across process boundaries, so you may wish to emulate these features in a way that makes it easier to accomplish these other tasks. From the point of view of efficiency, however, you are unlikely to do better than the

compiler-generated implementations by coding these features yourself.

Back to Item 23: Consider alternative libraries
Continue to Techniques

Back to Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI
Continue to Item 25: Virtualizing constructors and non-member functions

Techniques

Most of this book is concerned with programming guidelines. Such guidelines are important, but no programmer lives by guidelines alone. According to the old TV show *Felix the Cat*, "Whenever he gets in a fix, he reaches into his bag of tricks." Well, if a cartoon character can have a bag of tricks, so too can C++ programmers. Think of this chapter as a starter set for your bag of tricks.

Some problems crop up repeatedly when designing C++ software. How can you make constructors and non-member functions act like virtual functions? How can you limit the number of instances of a class? How can you prevent objects from being created on the heap? How can you guarantee that they will be created there? How can you create objects that automatically perform some actions anytime some other class's member functions are called? How can you have different objects share data structures while giving clients the illusion that each has its own copy? How can you distinguish between read and write usage of `operator[]`? How can you create a virtual function whose behavior depends on the dynamic types of more than one object?

All these questions (and more) are answered in this chapter, in which I describe proven solutions to problems commonly encountered by C++ programmers. I call such solutions *techniques*, but they're also known as *idioms* and, when documented in a stylized fashion, *patterns*. Regardless of what you call them, the information that follows will serve you well as you engage in the day-to-day skirmishes of practical software development. It should also convince you that no matter what you want to do, there is almost certainly a way to do it in C++.

Back to Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI
Continue to Item 25: Virtualizing constructors and non-member functions

Back to Techniques
Continue to Item 26: Limiting the number of objects of a class

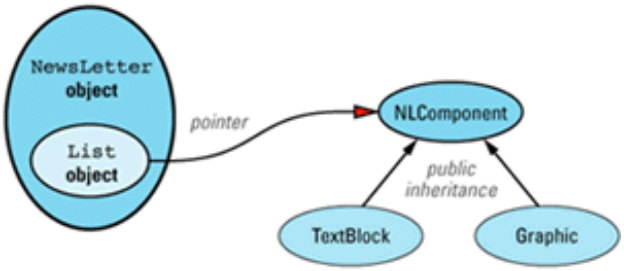
Item 25: Virtualizing constructors and non-member functions.

On the face of it, it doesn't make much sense to talk about "virtual constructors." You call a virtual function to achieve type-specific behavior when you have a pointer or reference to an object but you don't know what the real type of the object is. You call a constructor only when you don't yet have an object but you know exactly what type you'd like to have. How, then, can one talk of *virtual* constructors?

It's easy. Though virtual constructors may seem nonsensical, they are remarkably useful. (If you think nonsensical ideas are never useful, how do you explain the success of modern physics?) For example, suppose you write applications for working with newsletters, where a newsletter consists of components that are either textual or graphical. You might organize things this way:

```
class NLComponent {               // abstract base class for
public:                           // newsletter components

  ...                             // contains at least one
};                                // pure virtual function

class TextBlock: public NLComponent {
public:
  ...                             // contains no pure virtual
};                                // functions

class Graphic: public NLComponent {
public:
  ...                             // contains no pure virtual
};                                // functions

class NewsLetter {                // a newsletter object
public:                           // consists of a list of
  ...                             // NLComponent objects

private:
  list<NLComponent*> components;
};
```

The classes relate in this way:



The `list` class used inside `NewsLetter` is part of the Standard Template Library, which is part of the standard C++ library (see Item E49 and Item 35). Objects of type `list` behave like doubly linked lists, though they need not be implemented in that way.

`NewsLetter` objects, when not being worked on, would likely be stored on disk. To support the creation of a `Newsletter` from its on-disk representation, it would be convenient to give `NewsLetter` a constructor that takes an `istream`. The constructor would read information from the stream as it created the necessary in-core data structures:

```
class NewsLetter {
public:
  NewsLetter(istream& str);
  ...
};
```

Pseudocode for this constructor might look like this,

```
NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    read the next component object from str;

    add the object to the list of this
    newsletter's components;
  }
}
```

or, after moving the tricky stuff into a separate function called `readComponent`, like this:

```
class NewsLetter {
```

```
public:
  ...

private:
  // read the data for the next NLComponent from str,
  // create the component and return a pointer to it
  static NLComponent * readComponent(istream& str);
  ...
};

NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    // add the pointer returned by readComponent to the
    // end of the components list; "push_back" is a list
    // member function that inserts at the end of the list
    components.push_back(readComponent(str));
  }
}
```

Consider what `readComponent` does. It creates a new object, either a `TextBlock` or a `Graphic`, depending on the data it reads. Because it creates new objects, it acts much like a constructor, but because it can create different types of objects, we call it a *virtual constructor.* A virtual constructor is a function that creates different types of objects depending on the input it is given. Virtual constructors are useful in many contexts, only one of which is reading object information from disk (or off a network connection or from a tape, etc.).

A particular kind of virtual constructor — the *virtual copy constructor* — is also widely useful. A virtual copy constructor returns a pointer to a new copy of the object invoking the function. Because of this behavior, virtual copy constructors are typically given names like `copySelf`, `cloneSelf`, or, as shown below, just plain `clone`. Few functions are implemented in a more straightforward manner:

```
class NLComponent {
public:
  // declaration of virtual copy constructor
  virtual NLComponent * clone() const = 0;
  ...

};

class TextBlock: public NLComponent {
public:
  virtual TextBlock * clone() const         // virtual copy
  { return new TextBlock(*this); }          // constructor
  ...

};

class Graphic: public NLComponent {
```

```
public:
  virtual Graphic * clone() const            // virtual copy
  { return new Graphic(*this); }             // constructor
  ...

};
```

As you can see, a class's virtual copy constructor just calls its real copy constructor. The meaning of "copy" is hence the same for both functions. If the real copy constructor performs a shallow copy, so does the virtual copy constructor. If the real copy constructor performs a deep copy, so does the virtual copy constructor. If the real copy constructor does something fancy like reference counting or copy-on-write (see Item 29), so does the virtual copy constructor. Consistency — what a wonderful thing.

Notice that the above implementation takes advantage of a relaxation in the rules for virtual function return types that was adopted relatively recently. No longer must a derived class's redefinition of a base class's virtual function declare the same return type. Instead, if the function's return type is a pointer (or a reference) to a base class, the derived class's function may return a pointer (or reference) to a class derived from that base class. This opens no holes in C++'s type system, and it makes it possible to accurately declare functions such as virtual copy constructors. That's why `TextBlock`'s `clone` can return a `TextBlock*` and `Graphic`'s `clone` can return a `Graphic*`, even though the return type of `NLComponent`'s `clone` is `NLComponent*`.

The existence of a virtual copy constructor in `NLComponent` makes it easy to implement a (normal) copy constructor for `NewsLetter`:

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter& rhs);
  ...

private:
  list<NLComponent*> components;
};

NewsLetter::NewsLetter(const NewsLetter& rhs)
{
  // iterate over rhs's list, using each element's
  // virtual copy constructor to copy the element into
  // the components list for this object. For details on
  // how the following code works, see Item 35.
  for (list<NLComponent*>::const_iterator it =
          rhs.components.begin();
       it != rhs.components.end();
       ++it) {

    // "it" points to the current element of rhs.components,
    // so call that element's clone function to get a copy
```

```
    // of the element, and add that copy to the end of
    // this object's list of components
    components.push_back((*it)->clone());
  }
}
```

Unless you are familiar with the Standard Template Library, this code looks bizarre, I know, but the idea is simple: just iterate over the list of components for the `NewsLetter` object being copied, and for each component in the list, call its virtual copy constructor. We need a virtual copy constructor here, because the list contains pointers to `NLComponent` objects, but we know each pointer really points to a `TextBlock` or a `Graphic`. We want to copy whatever the pointer really points to, and the virtual copy constructor does that for us.

Making Non-Member Functions Act Virtual

Just as constructors can't really be virtual, neither can non-member functions (see Item E19). However, just as it makes sense to conceive of functions that construct new objects of different types, it makes sense to conceive of non-member functions whose behavior depends on the dynamic types of their parameters. For example, suppose you'd like to implement output operators for the `TextBlock` and `Graphic` classes. The obvious approach to this problem is to make the output operator virtual. However, the output operator is `operator<<`, and that function takes an `ostream&` as its left-hand argument; that effectively rules out the possibility of making it a member function of the `TextBlock` or `Graphic` classes.

(It can be done, but then look what happens:

```
class NLComponent {
public:
  // unconventional declaration of output operator
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  // virtual output operator (also unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

class Graphic: public NLComponent {
public:
  // virtual output operator (still unconventional)
  virtual ostream& operator<<(ostream& str) const;
};

TextBlock t;
Graphic g;
```

```
...

t << cout;                                  // print t on cout via
                                            // virtual operator<<; note
                                            // unconventional syntax

g << cout;                                  // print g on cout via
                                            // virtual operator<<; note
                                            // unconventional syntax
```

Clients must place the stream object on the *right-hand side* of the "<<" symbol, and that's contrary to the convention for output operators. To get back to the normal syntax, we must move `operator<<` out of the `TextBlock` and `Graphic` classes, but if we do that, we can no longer declare it virtual.)

An alternate approach is to declare a virtual function for printing (e.g., `print`) and define it for the `TextBlock` and `Graphic` classes. But if we do that, the syntax for printing `TextBlock` and `Graphic` objects is inconsistent with that for the other types in the language, all of which rely on `operator<<` as their output operator.

Neither of these solutions is very satisfying. What we want is a non-member function called `operator<<` that exhibits the behavior of a virtual function like `print`. This description of what we want is in fact very close to a description of how to get it. We define *both* `operator<<` and `print` and have the former call the latter!

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...

};

class TextBlock: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...

};

inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
```

```
  return c.print(s);
}
```

Virtual-acting non-member functions, then, are easy. You write virtual functions to do the work, then write a non-virtual function that does nothing but call the virtual function. To avoid incurring the cost of a function call for this syntactic sleight-of-hand, of course, you inline the non-virtual function (see Item E33).

Now that you know how to make non-member functions act virtually on one of their arguments, you may wonder if it's possible to make them act virtually on more than one of their arguments. It is, but it's not easy. How hard is it? Turn to Item 31; it's devoted to that question.

Back to Techniques
Continue to Item 26: Limiting the number of objects of a class

Item 26: Limiting the number of objects of a class.

Okay, you're crazy about objects, but sometimes you'd like to bound your insanity. For example, you've got only one printer in your system, so you'd like to somehow limit the number of printer objects to one. Or you've got only 16 file descriptors you can hand out, so you've got to make sure there are never more than that many file descriptor objects in existence. How can you do such things? How can you limit the number of objects?

If this were a proof by mathematical induction, we might start with n = 1, then build from there. Fortunately, this is neither a proof nor an induction. Moreover, it turns out to be instructive to begin with n = 0, so we'll start there instead. How do you prevent objects from being instantiated at all?

Allowing Zero or One Objects

Each time an object is instantiated, we know one thing for sure: a constructor will be called. That being the case, the easiest way to prevent objects of a particular class from being created is to declare the constructors of that class private:

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);

  ...

};
```

Having thus removed everybody's right to create objects, we can selectively loosen the restriction. If, for example, we want to create a class for printers, but we also want to abide by the constraint that there is only one printer available to us, we can encapsulate the printer object inside a function so that everybody has access to the printer, but only a single printer object is created:

```
class PrintJob;                                  // forward declaration
                                                 // see Item E34

class Printer {
public:
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();

  ...

friend Printer& thePrinter();

private:
```

```
  Printer();
  Printer(const Printer& rhs);

  ...

};

Printer& thePrinter()
{
  static Printer p;                          // the single printer object
  return p;
}
```

There are three separate components to this design. First, the constructors of the `Printer` class are private. That suppresses object creation. Second, the global function `thePrinter` is declared a friend of the class. That lets `thePrinter` escape the restriction imposed by the private constructors. Finally, `thePrinter` contains a *static* `Printer` object. That means only a single object will be created.

Client code refers to `thePrinter` whenever it wishes to interact with the system's lone printer. By returning a reference to a `Printer` object, `thePrinter` can be used in any context where a `Printer` object itself could be:

```
class PrintJob {
public:
  PrintJob(const string& whatToPrint);
  ...

};

string buffer;

...                                          // put stuff in buffer

thePrinter().reset();
thePrinter().submitJob(buffer);
```

It's possible, of course, that `thePrinter` strikes you as a needless addition to the global namespace. "Yes," you may say, "as a global function it looks more like a global variable, but global variables are gauche, and I'd prefer to localize all printer-related functionality inside the `Printer` class." Well, far be it from me to argue with someone who uses words like gauche. `thePrinter` can just as easily be made a static member function of `Printer` , and that puts it right where you want it. It also eliminates the need for a `friend` declaration, which many regard as tacky in its own right. Using a static member function, `Printer` looks like this:

```
class Printer {
public:
  static Printer& thePrinter();
  ...

private:
```

```
  Printer();
  Printer(const Printer& rhs);
  ...

};

Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

Clients must now be a bit wordier when they refer to the printer:

```
Printer::thePrinter().reset();
Printer::thePrinter().submitJob(buffer);
```

Another approach is to move `Printer` and `thePrinter` out of the global scope and into a *namespace* (see Item E28). Namespaces are a recent addition to C++. Anything that can be declared at global scope can also be declared in a namespace. This includes classes, structs, functions, variables, objects, typedefs, etc. The fact that something is in a namespace doesn't affect its behavior, but it does prevent name conflicts between entities in different namespaces. By putting the `Printer` class and the `thePrinter` function into a namespace, we don't have to worry about whether anybody else happened to choose the names `Printer` or `thePrinter` for themselves; our namespace prevents name conflicts.

Syntactically, namespaces look much like classes, but there are no public, protected, or private sections; everything is public. This is how we'd put `Printer` and `thePrinter` into a namespace called `PrintingStuff` :

```
namespace PrintingStuff {
  class Printer {                          // this class is in the
  public:                                  // PrintingStuff namespace

    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...

    friend Printer& thePrinter();

  private:
    Printer();
    Printer(const Printer& rhs);
    ...

 };

 Printer& thePrinter()                      // so is this function
 {
    static Printer p;
    return p;
```

```
 }
}                                                  // this is the end of the
                                                   // namespace
```

Given this namespace, clients can refer to `thePrinter` using a fully-qualified name (i.e., one that includes the name of the namespace),

```
PrintingStuff::thePrinter().reset();
PrintingStuff::thePrinter().submitJob(buffer);
```

but they can also employ a *`using` declaration* to save themselves keystrokes:

```
using PrintingStuff::thePrinter;                   // import the name
                                                   // "thePrinter" from the
                                                   // namespace "PrintingStuff"
                                                   // into the current scope


thePrinter().reset();                              // now thePrinter can be
thePrinter().submitJob(buffer);                    // used as if it were a
                                                   // local name
```

There are two subtleties in the implementation of `thePrinter` that are worth exploring. First, it's important that the single `Printer` object be static in a *function* and not in a class. An object that's static in a class is, for all intents and purposes, *always* constructed (and destructed), even if it's never used. In contrast, an object that's static in a function is created the first time through the function, so if the function is never called, the object is never created. (You do, however, pay for a check each time the function is called to see whether the object needs to be created.) One of the philosophical pillars on which C++ was built is the idea that you shouldn't pay for things you don't use, and defining an object like our printer as a static object in a function is one way of adhering to this philosophy. It's a philosophy you should adhere to whenever you can.

There is another drawback to making the printer a class static versus a function static, and that has to do with its time of initialization. We know exactly when a function static is initialized: the first time through the function at the point where the static is defined. The situ ation with a class static (or, for that matter, a global static, should you be so gauche as to use one) is less well defined. C++ offers certain guarantees regarding the order of initialization of statics within a particular translation unit (i.e., a body of source code that yields a single object file), but it says *nothing* about the initialization order of static objects in different translation units (see Item E47). In practice, this turns out to be a source of countless headaches. Function statics, when they can be made to suffice, allow us to avoid these headaches. In our example here, they can, so why suffer?

The second subtlety has to do with the interaction of inlining and static objects inside functions. Look again at the code for the non-member version of `thePrinter` :

```
Printer& thePrinter()
{
  static Printer p;
```

```
  return p;
}
```

Except for the first time through this function (when `p` must be constructed), this is a one-line function — it consists entirely of the statement "`return p ;`". If ever there were a good candidate for inlining, this function would certainly seem to be the one. Yet it's not declared `inline` . Why not?

Consider for a moment why you'd declare an object to be static. It's usually because you want only a single copy of that object, right? Now consider what `inline` means. Conceptually, it means compilers should replace each call to the function with a copy of the function body, but for non-member functions, it also means something else. It means the functions in question have *internal linkage* .

You don't ordinarily need to worry about such linguistic mumbo jumbo, but there is one thing you must remember: functions with internal linkage may be duplicated within a program (i.e., the object code for the program may contain more than one copy of each function with internal linkage), and *this duplication includes static objects contained within the functions* . The result? If you create an inline non-member function containing a local static object, you may end up with *more than one copy* of the static object in your program! So don't create inline non-member functions that contain local static data.[9]

But maybe you think this business of creating a function to return a reference to a hidden object is the wrong way to go about limiting the number of objects in the first place. Perhaps you think it's better to simply count the number of objects in existence and throw an excep tion in a constructor if too many objects are requested. In other words, maybe you think we should handle printer creation like this:

```
class Printer {
public:
  class TooManyObjects{};                    // exception class for use
                                             // when too many objects
                                             // are requested
  Printer();
  ~Printer();

  ...

private:
  static size_t numObjects;

  Printer(const Printer& rhs);               // there is a limit of 1
                                             // printer, so never allow
};                                           // copying (see Item E27)
```

The idea is to use `numObjects` to keep track of how many `Printer` objects are in existence. This value will be incremented in the class constructor and decremented in its destructor. If an attempt is made to construct too many Printer objects, we throw an exception of type `TooManyObjects` :

```
// Obligatory definition of the class static
```

```
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal construction here;

  ++numObjects;
}

Printer::~Printer()
{
  perform normal destruction here;

  --numObjects;
}
```

This approach to limiting object creation is attractive for a couple of reasons. For one thing, it's straightforward — everybody should be able to understand what's going on. For another, it's easy to generalize so that the maximum number of objects is some number other than one.

Contexts for Object Construction

There is also a problem with this strategy. Suppose we have a special kind of printer, say, a color printer. The class for such printers would have much in common with our generic printer class, so of course we'd inherit from it:

```
class ColorPrinter: public Printer {
  ...

};
```

Now suppose we have one generic printer and one color printer in our system:

```
Printer p;
ColorPrinter cp;
```

How many `Printer` objects result from these object definitions? The answer is two: one for `p` and one for the `Printer` part of `cp` . At runtime, a `TooManyObjects` exception will be thrown during the construction of the base class part of `cp` . For many programmers, this is neither what they want nor what they expect. (Designs that avoid having concrete classes inherit from other concrete classes do not suffer from this problem. For details on this design philosophy, see Item 33 .)

A similar problem occurs when `Printer` objects are contained inside other objects:

```
class CPFMachine {                          // for machines that can
```

```
private:                                    // copy, print, and fax

  Printer p;                                // for printing capabilities
  FaxMachine f;                             // for faxing capabilities
  CopyMachine c;                            // for copying capabilities

  ...

};

CPFMachine m1;                              // fine

CPFMachine m2;                              // throws TooManyObjects exception
```

The problem is that `Printer` objects can exist in three different contexts: on their own, as base class parts of more derived objects, and embedded inside larger objects. The presence of these different contexts significantly muddies the waters regarding what it means to keep track of the "number of objects in existence," because what you consider to be the existence of an object may not jibe with your compilers'.

Often you will be interested only in allowing objects to exist on their own, and you will wish to limit the number of *those* kinds of instantiations. That restriction is easy to satisfy if you adopt the strategy exem plified by our original `Printer` class, because the `Printer` constructors are private, and (in the absence of `friend` declarations) classes with private constructors can't be used as base classes, nor can they be embedded inside other objects.

The fact that you can't derive from classes with private constructors leads to a general scheme for preventing derivation, one that doesn't necessarily have to be coupled with limiting object instantiations. Suppose, for example, you have a class, `FSA` , for representing finite state automata. (Such state machines are useful in many contexts, among them user interface design.) Further suppose you'd like to allow any number of `FSA` objects to be created, but you'd also like to ensure that no class ever inherits from `FSA` . (One reason for doing this might be to justify the presence of a nonvirtual destructor in `FSA` . Item E14 explains why base classes generally need virtual destructors, and Item 24 explains why classes without virtual functions yield smaller objects than do equivalent classes with virtual functions.) Here's how you can design `FSA` to satisfy both criteria:

```
class FSA {
public:
  // pseudo-constructors
  static FSA * makeFSA();
  static FSA * makeFSA(const FSA& rhs);
  ...

private:
  FSA();
  FSA(const FSA& rhs);
  ...
};

FSA * FSA::makeFSA()
{ return new FSA(); }
```

```
FSA * FSA::makeFSA(const FSA& rhs)
{ return new FSA(rhs); }
```

Unlike the `thePrinter` function that always returned a reference to a single object, each `makeFSA` pseudo-constructor returns a pointer to a unique object. That's what allows an unlimited number of `FSA` objects to be created.

This is nice, but the fact that each pseudo-constructor calls `new` implies that callers will have to remember to call `delete`. Otherwise a resource leak will be introduced. Callers who wish to have `delete` called automatically when the current scope is exited can store the pointer returned from `makeFSA` in an `auto_ptr` object (see Item 9); such objects automatically delete what they point to when they themselves go out of scope:

```
// indirectly call default FSA constructor
auto_ptr<FSA> pfsa1(FSA::makeFSA());

// indirectly call FSA copy constructor
auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));

...                             // use pfsa1 and pfsa2 as normal pointers,
                                // but don't worry about deleting them
```

Allowing Objects to Come and Go

We now know how to design a class that allows only a single instantiation, we know that keeping track of the number of objects of a particular class is complicated by the fact that object constructors are called in three different contexts, and we know that we can eliminate the confusion surrounding object counts by making constructors private. It is worthwhile to make one final observation. Our use of the `thePrinter` function to encapsulate access to a single object limits the number of `Printer` objects to one, but it also limits us to a single `Printer` object for each run of the program. As a result, it's not possible to write code like this:

```
create Printer object p1;

use p1;

destroy p1;

create Printer object p2;

use p2;

destroy p2;

...
```

This design never instantiates more than a single `Printer` object at a time, but it does use different `Printer` objects in different parts of the program. It somehow seems unreasonable that this isn't

allowed. After all, at no point do we violate the constraint that only one printer may exist. Isn't there a way to make this legal?

There is. All we have to do is combine the object-counting code we used earlier with the pseudo-constructors we just saw:

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructor
  static Printer * makePrinter();

~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

private:
  static size_t numObjects;

  Printer();

  Printer(const Printer& rhs);          // we don't define this
};                                      // function, because we'll
                                        // never allow copying
                                        // (see Item E27)

// Obligatory definition of class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal object construction here;

  ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

If the notion of throwing an exception when too many objects are requested strikes you as unreasonably harsh, you could have the pseudo-constructor return a null pointer instead. Clients would then have to check for this before doing anything with it, of course.

Clients use this `Printer` class just as they would any other class, except they must call the pseudo-constructor function instead of the real constructor:

```
Printer p1;                                  // error! default ctor is
                                             // private

Printer *p2 =
  Printer::makePrinter();                    // fine, indirectly calls
                                             // default ctor

Printer p3 = *p2;                            // error! copy ctor is
                                             // private

p2->performSelfTest();                       // all other functions are
p2->reset();                                 // called as usual

...

delete p2;                                   // avoid resource leak; this
                                             // would be unnecessary if
                                             // p2 were an auto_ptr
```

This technique is easily generalized to any number of objects. All we have to do is replace the hard-wired constant 1 with a class-specific value, then lift the restriction against copying objects. For example, the following revised implementation of our `Printer` class allows up to 10 `Printer` objects to exist:

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ...

private:
  static size_t numObjects;
  static const size_t maxObjects = 10;       // see below

  Printer();
  Printer(const Printer& rhs);
};

// Obligatory definitions of class statics
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;

Printer::Printer()
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...
```

```
}

Printer::Printer(const Printer& rhs)
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...

}

Printer * Printer::makePrinter()
{ return new Printer; }

Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

Don't be surprised if your compilers get all upset about the declaration of `Printer::maxObjects` in the class definition above. In particular, be prepared for them to complain about the specification of 10 as an initial value for that variable. The ability to specify initial values for static `const` members (of integral type, e.g., `int` s, `char` s, enums, etc.) inside a class definition was added to C++ only relatively recently, so some compilers don't yet allow it. If your compilers are as-yet-unupdated, pacify them by declaring `maxObjects` to be an enumerator inside a private anonymous enum,

```
class Printer {
private:
  enum { maxObjects = 10 };                 // within this class,
  ...                                       // maxObjects is the
};                                          // constant 10
```

or by initializing the constant static like a non-`const` static member:

```
class Printer {
private:
  static const size_t maxObjects;           // no initial value given

  ...

};

// this goes in a single implementation file
const size_t Printer::maxObjects = 10;
```

This latter approach has the same effect as the original code above, but explicitly specifying the initial value is easier for other programmers to understand. When your compilers support the specification of initial values for `const` static members in class definitions, you should take advantage of that capability.

An Object-Counting Base Class

Initialization of statics aside, the approach above works like the proverbial charm, but there is one aspect of it that continues to nag. If we had a lot of classes like `Printer` whose instantiations needed to be limited, we'd have to write this same code over and over, once per class. That would be mind-numbingly dull. Given a fancy-pants language like C++, it somehow seems we should be able to automate the process. Isn't there a way to encapsulate the notion of counting instances and bundle it into a class?

We can easily come up with a base class for counting object instances and have classes like `Printer` inherit from that, but it turns out we can do even better. We can actually come up with a way to encapsulate the whole counting kit and kaboodle, by which I mean not only the functions to manipulate the instance count, but also the instance count itself. (We'll see the need for a similar trick when we examine reference counting in Item 29 . For a detailed examination of this design, see my article on counting objects.)

The counter in the `Printer` class is the static variable `numObjects` , so we need to move that variable into an instance-counting class. However, we also need to make sure that each class for which we're counting instances has a *separate* counter. Use of a counting class *template* lets us automatically generate the appropriate number of counters, because we can make the counter a static member of the classes generated from the template:

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};                     // for throwing exceptions

  static int objectCount() { return numObjects; }

protected:
  Counted();
  Counted(const Counted& rhs);

  ~Counted() { --numObjects; }

private:
  static int numObjects;
  static const size_t maxObjects;

  void init();                                // to avoid ctor code
};                                            // duplication

template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }

template<class BeingCounted>
Counted<BeingCounted>::Counted(const Counted<BeingCounted>&)
{ init(); }

template<class BeingCounted>
void Counted<BeingCounted>::init()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
```

```
  ++numObjects;
}
```

The classes generated from this template are designed to be used only as base classes, hence the protected constructors and destructor. Note the use of the private member function `init` to avoid duplicating the statements in the two `Counted` constructors.

We can now modify the `Printer` class to use the `Counted` template:

```
class Printer: private Counted<Printer> {
public:
  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

  using Counted<Printer>::objectCount;     // see below
  using Counted<Printer>::TooManyObjects;  // see below

private:
  Printer();
  Printer(const Printer& rhs);
};
```

The fact that `Printer` uses the `Counted` template to keep track of how many `Printer` objects exist is, frankly, nobody's business but the author of `Printer` 's. Such implementation details are best kept private, and that's why private inheritance is used here (see Item E42). The alternative would be to use public inheritance between `Printer` and `Counted<Printer>` , but then we'd be obliged to give the `Counted` classes a virtual destructor. (Otherwise we'd risk incorrect behavior if somebody deleted a `Printer` object through a `Counted<Printer>*` pointer — see Item E14 .) As Item 24 makes clear, the presence of a virtual function in `Counted` would almost certainly affect the size and layout of objects of classes inheriting from `Counted` . We don't want to absorb that overhead, and the use of private inheritance lets us avoid it.

Quite properly, most of what `Counted` does is hidden from `Printer` 's clients, but those clients might reasonably want to find out how many `Printer` objects exist. The `Counted` template offers the `objectCount` function to provide this information, but that function becomes private in `Printer` due to our use of private inheritance. To restore the public accessibility of that function, we employ a `using` declaration:

```
class Printer: private Counted<Printer> {
public:
  ...
  using Counted<Printer>::objectCount; // make this function
```

```
                                        // public for clients
  ...                                   // of Printer
};
```

This is perfectly legitimate, but if your compilers don't yet support namespaces, they won't allow it. If they don't, you can use the older access declaration syntax:

```
class Printer: private Counted<Printer> {
public:
  ...
  Counted<Printer>::objectCount;        // make objectCount
                                        // public in Printer
  ...

};
```

This more traditional syntax has the same meaning as the `using` declaration, but it's deprecated. The class `TooManyObjects` is handled in the same fashion as `objectCount` , because clients of `Printer` must have access to `TooManyObjects` if they are to be able to catch exceptions of that type.

When `Printer` inherits from `Counted<Printer>` , it can forget about counting objects. The class can be written as if somebody else were doing the counting for it, because somebody else (`Counted<Printer>`) is. A `Printer` constructor now looks like this:

```
Printer::Printer()
{
  proceed with normal object construction;
}
```

What's interesting here is not what you see, it's what you don't. No checking of the number of objects to see if the limit is about to be exceeded, no incrementing the number of objects in existence once the constructor is done. All that is now handled by the `Counted<Printer>` constructors, and because `Counted<Printer>` is a base class of `Printer` , we know that a `Counted<Printer>` constructor will always be called before a `Printer` constructor. If too many objects are created, a `Counted<Printer>` constructor throws an exception, and the `Printer` constructor won't even be invoked. Nifty, huh?

Nifty or not, there's one loose end that demands to be tied, and that's the mandatory definitions of the statics inside `Counted` . It's easy enough to take care of `numObjects` — we just put this in `Counted` 's implementation file:

```
template<class BeingCounted>                // defines numObjects
int Counted<BeingCounted>::numObjects;      // and automatically
                                            // initializes it to 0
```

The situation with `maxObjects` is a bit trickier. To what value should we initialize this variable? If we want to allow up to 10 printers, we should initialize `Counted<Printer>` ::`maxObjects` to 10. If,

on the other hand, we want to allow up to 16 file descriptor objects, we should initialize `Counted<FileDescriptor>::maxObjects` to 16. What to do?

We take the easy way out: we do nothing. We provide no initialization at all for `maxObjects` . Instead, we require that *clients* of the class provide the appropriate initialization. The author of `Printer` must add this to an implementation file:

```
const size_t Counted<Printer>::maxObjects = 10;
```

Similarly, the author of `FileDescriptor` must add this:

```
const size_t Counted<FileDescriptor>::maxObjects = 16;
```

What will happen if these authors forget to provide a suitable definition for `maxObjects` ? Simple: they'll get an error during linking, because `maxObjects` will be undefined. Provided we've adequately documented this requirement for clients of `Counted` , they can then say "Duh" to themselves and go back and add the requisite initialization.

Back to Item 25: Virtualizing constructors and non-member functions
Continue to Item 27: Requiring or prohibiting heap-based objects

---

[9] In July 1996, the ◦ ISO/ANSI standardization committee changed the default linkage of inline functions to *external* , so the problem I describe here has been eliminated, at least on paper. Your compilers may not yet be in accord with ◦ the standard , however, so your best bet is still to shy away from inline functions with static data.
Return

Back to Item 26: Limiting the number of objects of a class
Continue to Item 28: Smart pointers

Item 27: Requiring or prohibiting heap-based objects.

Sometimes you want to arrange things so that objects of a particular type can commit suicide, i.e., can "`delete this`." Such an arrangement clearly requires that objects of that type be allocated on the heap. Other times you'll want to bask in the certainty that there can be no memory leaks for a particular class, because none of the objects could have been allocated on the heap. This might be the case if you are working on an embedded system, where memory leaks are especially troublesome and heap space is at a premium. Is it possible to produce code that requires or prohibits heap-based objects? Often it is, but it also turns out that the notion of being "on the heap" is more nebulous than you might think.

Requiring Heap-Based Objects

Let us begin with the prospect of limiting object creation to the heap. To enforce such a restriction, you've got to find a way to prevent clients from creating objects other than by calling `new`. This is easy to do. Non-heap objects are automatically constructed at their point of definition and automatically destructed at the end of their lifetime, so it suffices to simply make these implicit constructions and destructions illegal.

The straightforward way to make these calls illegal is to declare the constructors and the destructor `private`. This is overkill. There's no reason why they *both* need to be private. Better to make the destructor private and the constructors public. Then, in a process that should be familiar from Item 26, you can introduce a privileged pseudo-destructor function that has access to the real destructor. Clients then call the pseudo-destructor to destroy the objects they've created.

If, for example, we want to ensure that objects representing unlimited precision numbers are created only on the heap, we can do it like this:

```
class UPNumber {
public:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);

  // pseudo-destructor (a const member function, because
  // even const objects may be destroyed)
  void destroy() const { delete this; }

  ...

private:
  ~UPNumber();
```

```
};
```

Clients would then program like this:

```
UPNumber n;                          // error! (legal here, but
                                     // illegal when n's dtor is
                                     // later implicitly invoked)

UPNumber *p = new UPNumber;          // fine

...

delete p;                            // error! attempt to call
                                     // private destructor

p->destroy();                        // fine
```

An alternative is to declare all the constructors private. The drawback to that idea is that a class often has many constructors, and the class's author must remember to declare each of them private. This includes the copy constructor, and it may include a default constructor, too, if these functions would otherwise be generated by compilers; compiler-generated functions are always public (see Item E45). As a result, it's easier to declare only the destructor private, because a class can have only one of those.

Restricting access to a class's destructor or its constructors prevents the creation of non-heap objects, but, in a story that is told in Item 26, it also prevents both inheritance and containment:

```
class UPNumber { ... };              // declares dtor or ctors
                                     // private

class NonNegativeUPNumber:
  public UPNumber { ... };           // error! dtor or ctors
                                     // won't compile

class Asset {
private:
  UPNumber value;
  ...                                // error! dtor or ctors
                                     // won't compile
};
```

Neither of these difficulties is insurmountable. The inheritance problem can be solved by making `UPNumber`'s destructor protected (while keeping its constructors public), and classes that need to contain objects of type `UPNumber` can be modified to contain *pointers* to `UPNumber` objects instead:

```
class UPNumber { ... };              // declares dtor protected

class NonNegativeUPNumber:
  public UPNumber { ... };           // now okay; derived
                                     // classes have access to
                                     // protected members

class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...

private:
  UPNumber *value;
};

Asset::Asset(int initValue)
: value(new UPNumber(initValue))      // fine
{ ... }

Asset::~Asset()
{ value->destroy(); }                 // also fine
```

Determining Whether an Object is On The Heap

If we adopt this strategy, we must reexamine what it means to be "on the heap." Given the class definition sketched above, it's legal to define a non-heap `NonNegativeUPNumber` object:

```
NonNegativeUPNumber n;                // fine
```

Now, the `UPNumber` part of the `NonNegativeUPNumber` object `n` is not on the heap. Is that okay? The answer depends on the details of the class's design and implementation, but let us suppose it is *not* okay, that all `UPNumber` objects — even base class parts of more derived objects — *must* be on the heap. How can we enforce this restriction?

There is no easy way. It is not possible for a `UPNumber` constructor to determine whether it's being invoked as the base class part of a heap-based object. That is, there is no way for the `UPNumber` constructor to detect that the following contexts are different:

```
NonNegativeUPNumber *n1 =
  new NonNegativeUPNumber;            // on heap

NonNegativeUPNumber n2;               // not on heap
```

But perhaps you don't believe me. Perhaps you think you can play games with the interaction

among the new operator, operator new and the constructor that the new operator calls (see Item 8). Perhaps you think you can outsmart them all by modifying UPNumber as follows:

```
class UPNumber {
public:
  // exception to throw if a non-heap object is created
  class HeapConstraintViolation {};

  static void * operator new(size_t size);

  UPNumber();
  ...

private:
  static bool onTheHeap;                // inside ctors, whether
                                        // the object being
  ...                                   // constructed is on heap

};

// obligatory definition of class static
bool UPNumber::onTheHeap = false;

void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;
  return ::operator new(size);
}

UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }

  proceed with normal construction here;

  onTheHeap = false;                    // clear flag for next obj.
}
```

There's nothing deep going on here. The idea is to take advantage of the fact that when an object is allocated on the heap, operator new is called to allocate the raw memory, then a constructor is called to initialize an object in that memory. In particular, operator new sets onTheHeap to true, and each constructor checks onTheHeap to see if the raw memory of the object being constructed was allocated by operator new. If not, an exception of type HeapConstraintViolation is thrown. Otherwise, construction proceeds as usual, and when construction is finished, onTheHeap is set to false, thus resetting the default value for the next object to be constructed.

This is a nice enough idea, but it won't work. Consider this potential client code:

```
UPNumber *numberArray = new UPNumber[100];
```

The first problem is that the memory for the array is allocated by `operator new[]`, not `operator new`, but (provided your compilers support it) you can write the former function as easily as the latter. What is more troublesome is the fact that `numberArray` has 100 elements, so there will be 100 constructor calls. But there is only one call to allocate memory, so `onTheHeap` will be set to true for only the first of those 100 constructors. When the second constructor is called, an exception is thrown, and woe is you.

Even without arrays, this bit-setting business may fail. Consider this statement:

```
UPNumber *pn = new UPNumber(*new UPNumber);
```

Here we create two `UPNumbers` on the heap and make `pn` point to one of them; it's initialized with the value of the second one. This code has a resource leak, but let us ignore that in favor of an examination of what happens during execution of this expression:

```
new UPNumber(*new UPNumber)
```

This contains two calls to the `new` operator, hence two calls to `operator new` and two calls to `UPNumber` constructors (see Item 8). Programmers typically expect these function calls to be executed in this order,

1. Call `operator new` for first object
2. Call constructor for first object
3. Call `operator new` for second object
4. Call constructor for second object

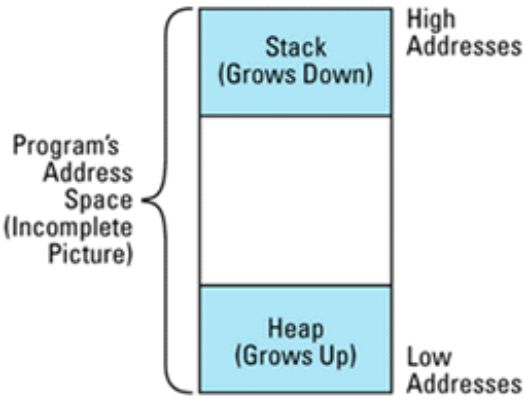
but the language makes no guarantee that this is how it will be done. Some compilers generate the function calls in this order instead:

1. Call `operator new` for first object
2. Call `operator new` for second object
3. Call constructor for first object
4. Call constructor for second object

There is nothing wrong with compilers that generate this kind of code, but the set-a-bit-in-`operator new` trick fails with such compilers. That's because the bit set in steps 1 and 2 is cleared in step 3, thus making the object constructed in step 4 think it's not on the heap, even though it is.

These difficulties don't invalidate the basic idea of having each constructor check to see if `*this` is on the heap. Rather, they indicate that checking a bit set inside `operator new` (or `operator new[]`) is not a reliable way to determine this information. What we need is a better way to figure it out.

If you're desperate enough, you might be tempted to descend into the realm of the unportable. For example, you might decide to take advantage of the fact that on many systems, a program's address space is organized as a linear sequence of addresses, with the program's stack growing down from the top of the address space and the heap rising up from the bottom:



On systems that organize a program's memory in this way (many do, but many do not), you might think you could use the following function to determine whether a particular address is on the heap:

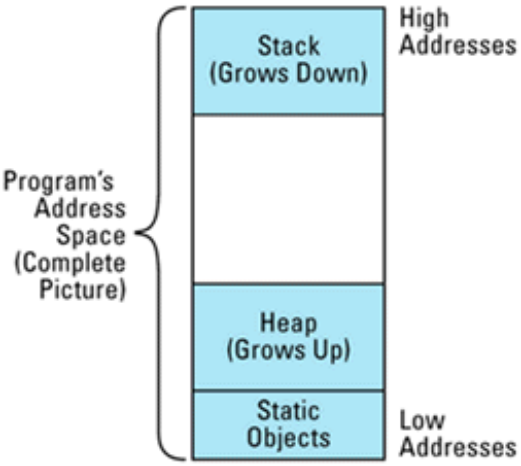
```
// incorrect attempt to determine whether an address
// is on the heap
bool onHeap(const void *address)
{
  char onTheStack;                  // local stack variable

  return address < &onTheStack;
}
```

The thinking behind this function is interesting. Inside `onHeap`, `onTheStack` is a local variable. As such, it is, well, it's on the stack. When `onHeap` is called, its stack frame (i.e., its activation record) will be placed at the top of the program's stack, and because the stack grows down (toward lower addresses) in this architecture, the address of `onTheStack` must be less than the address of any other

stack-based variable or object. If the parameter `address` is less than the location of `onTheStack`, it can't be on the stack, so it must be on the heap.

Such logic is fine, as far as it goes, but it doesn't go far enough. The fundamental problem is that there are *three* places where objects may be allocated, not two. Yes, the stack and the heap hold objects, but let us not forget about *static* objects. Static objects are those that are initialized only once during a program run. Static objects comprise not only those objects explicitly declared `static`, but also objects at global and namespace scope (see Item E47). Such objects have to go somewhere, and that somewhere is neither the stack nor the heap.

Where they go is system-dependent, but on many of the systems that have the stack and heap grow toward one another, they go below the heap. The earlier picture of memory organization, while telling the truth and nothing but the truth for many systems, failed to tell the whole truth for those systems. With static objects added to the picture, it looks like this:



Suddenly it becomes clear why `onHeap` won't work, not even on systems where it's purported to: it fails to distinguish between heap objects and static objects:

```
void allocateSomeObjects()
{
  char *pc = new char;              // heap object: onHeap(pc)
                                    // will return true
```

```
  char c;                              // stack object: onHeap(&c)
                                       // will return false

  static char sc;                      // static object: onHeap(&sc)
                                       // will return true
  ...

}
```

Now, you may be desperate for a way to tell heap objects from stack objects, and in your desperation you may be willing to strike a deal with the portability Devil, but are you so desperate that you'll strike a deal that fails to guarantee you the right answers? Surely not, so I know you'll reject this seductive but unreliable compare-the-addresses trick.

The sad fact is there's not only no portable way to determine whether an object is on the heap, there isn't even a semi-portable way that works most of the time. If you absolutely, positively have to tell whether an address is on the heap, you're going to have to turn to unportable, implementation-dependent system calls, and that's that. (It turns out that that may *not* be that. For details, consult the "Comments on M27" Web Page.) As such, you're better off trying to redesign your software so you don't need to determine whether an object is on the heap in the first place.

If you find yourself obsessing over whether an object is on the heap, the likely cause is that you want to know if it's safe to invoke `delete` on it. Often such deletion will take the form of the infamous "`delete this`." Knowing whether it's safe to delete a pointer, however, is not the same as simply knowing whether that pointer points to something on the heap, because not all pointers to things on the heap can be safely `delete`d. Consider again an `Asset` object that contains a `UPNumber` object:

```
class Asset {
private:
  UPNumber value;
  ...

};

Asset *pa = new Asset;
```

Clearly `*pa` (including its member `value`) is on the heap. Equally clearly, it's not safe to invoke `delete` on a pointer to `pa->value`, because no such pointer was ever returned from `new`.

As luck would have it, it's easier to determine whether it's safe to delete a pointer than to determine whether a pointer points to something on the heap, because all we need to answer the former question is a collection of addresses that have been returned by `operator new`. Since we can write

operator new ourselves (see Items E8-E10), it's easy to construct such a collection. Here's how we might approach the problem:

```
void *operator new(size_t size)
{
  void *p = getMemory(size);          // call some function to
                                      // allocate memory and
                                      // handle out-of-memory
                                      // conditions

  add p to the collection of allocated addresses;

  return p;

}

void operator delete(void *ptr)
{
  releaseMemory(ptr);                 // return memory to
                                      // free store

  remove ptr from the collection of allocated addresses;
}

bool isSafeToDelete(const void *address)
{
  return whether address is in collection of
  allocated addresses;
}
```

This is about as simple as it gets. operator new adds entries to a collection of allocated addresses, operator delete removes entries, and isSafeToDelete does a lookup in the collection to see if a particular address is there. If the operator new and operator delete functions are at global scope, this should work for all types, even the built-ins.

In practice, three things are likely to dampen our enthusiasm for this design. The first is our extreme reluctance to define anything at global scope, especially functions with predefined meanings like operator new and operator delete. Knowing as we do that there is but one global scope and but a single version of operator new and operator delete with the "normal" signatures (i.e., sets of parameter types) within that scope (see Item E9), the last thing we want to do is seize those function signatures for ourselves. Doing so would render our software incompatible with any other software that also implements global versions of operator new and operator delete (such as many object-oriented database systems).

Our second consideration is one of efficiency: why burden all heap allocations with the bookkeeping overhead necessary to keep track of returned addresses if we don't need to?

Our final concern is pedestrian, but important. It turns out to be essentially impossible to implement `isSafeToDelete` so that it always works. The difficulty has to do with the fact that objects with multiple or virtual base classes have multiple addresses, so there's no guarantee that the address passed to `isSafeToDelete` is the same as the one returned from `operator new`, even if the object in question was allocated on the heap. For details, see Items 24 and 31.

What we'd like is the functionality provided by these functions without the concomitant pollution of the global namespace, the mandatory overhead, and the correctness problems. Fortunately, C++ gives us exactly what we need in the form of an abstract mixin base class.

An abstract base class is a base class that can't be instantiated, i.e., one with at least one pure virtual function. A mixin ("mix in") class is one that provides a single well-defined capability and is designed to be compatible with any other capabilities an inheriting class might provide (see Item E7). Such classes are nearly always abstract. We can therefore come up with an abstract mixin base class that offers derived classes the ability to determine whether a pointer was allocated from `operator new`. Here's such a class:

```
class HeapTracked {                  // mixin class; keeps track of
public:                              // ptrs returned from op. new

  class MissingAddress{};            // exception class; see below

  virtual ~HeapTracked() = 0;

  static void *operator new(size_t size);
  static void operator delete(void *ptr);

  bool isOnHeap() const;

private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
```

This class uses the `list` data structure that's part of the standard C++ library (see Item E49 and Item 35) to keep track of all pointers returned from `operator new`. That function allocates memory and adds entries to the list; `operator delete` deallocates memory and removes entries from the list; and `isOnHeap` returns whether an object's address is in the list.

Implementation of the `HeapTracked` class is simple, because the global `operator new` and `operator delete` functions are called to perform the real memory allocation and deallocation, and the `list` class has functions to make insertion, removal, and lookup single-statement operations. Here's the full implementation of `HeapTracked`:

```
// mandatory definition of static class member
list<RawAddress> HeapTracked::addresses;

// HeapTracked's destructor is pure virtual to make the
// class abstract (see Item E14). The destructor must still
// be defined, however, so we provide this empty definition.
HeapTracked::~HeapTracked() {}



void * HeapTracked::operator new(size_t size)
{
  void *memPtr = ::operator new(size);  // get the memory

  addresses.push_front(memPtr);         // put its address at
                                        // the front of the list
  return memPtr;
}

void HeapTracked::operator delete(void *ptr)
{
  // get an "iterator" that identifies the list
  // entry containing ptr; see Item 35 for details
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), ptr);

  if (it != addresses.end()) {       // if an entry was found
    addresses.erase(it);             // remove the entry
    ::operator delete(ptr);          // deallocate the memory
  } else {                           // otherwise
    throw MissingAddress();          // ptr wasn't allocated by
  }                                  // op. new, so throw an
}                                    // exception

bool HeapTracked::isOnHeap() const
{
  // get a pointer to the beginning of the memory
  // occupied by *this; see below for details
  const void *rawAddress = dynamic_cast<const void*>(this);

  // look up the pointer in the list of addresses
  // returned by operator new
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), rawAddress);

  return it != addresses.end();      // return whether it was
}                                    // found
```

This code is straightforward, though it may not look that way if you are unfamiliar with the `list` class and the other components of the Standard Template Library. Item 35 explains everything, but the comments in the code above should be sufficient to explain what's happening in this example.

The only other thing that may confound you is this statement (in `isOnHeap`):

```
const void *rawAddress = dynamic_cast<const void*>(this);
```

I mentioned earlier that writing the global function `isSafeToDelete` is complicated by the fact that objects with multiple or virtual base classes have several addresses. That problem plagues us in `isOnHeap`, too, but because `isOnHeap` applies only to `HeapTracked` objects, we can exploit a special feature of the `dynamic_cast` operator (see Item 2) to eliminate the problem. Simply put, `dynamic_cast`ing a pointer to `void*` (or `const void*` or `volatile void*` or, for those who can't get enough modifiers in their usual diet, `const volatile void*`) yields a pointer to the beginning of the memory for the object pointed to by the pointer. But `dynamic_cast` is applicable only to pointers to objects that have at least one virtual function. Our ill-fated `isSafeToDelete` function had to work with *any* type of pointer, so `dynamic_cast` wouldn't help it. `isOnHeap` is more selective (it tests only pointers to `HeapTracked` objects), so `dynamic_cast`ing `this` to `const void*` gives us a pointer to the beginning of the memory for the current object. That's the pointer that `HeapTracked::operator new` must have returned if the memory for the current object was allocated by `HeapTracked::operator new` in the first place. Provided your compilers support the `dynamic_cast` operator, this technique is completely portable.

Given this class, even BASIC programmers could add to a class the ability to track pointers to heap allocations. All they'd need to do is have the class inherit from `HeapTracked`. If, for example, we want to be able to determine whether a pointer to an `Asset` object points to a heap-based object, we'd modify `Asset`'s class definition to specify `HeapTracked` as a base class:

```
class Asset: public HeapTracked {
private:
  UPNumber value;
  ...


};
```

We could then query `Asset*` pointers as follows:

```
void inventoryAsset(const Asset *ap)
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset — inventory it as such;
  }
  else {
    ap is a non-heap-based asset — record it that way;
  }
}
```

A disadvantage of a mixin class like `HeapTracked` is that it can't be used with the built-in types, because types like `int` and `char` can't inherit from anything. Still, the most common reason for wanting to use a class like `HeapTracked` is to determine whether it's okay to "`delete this`," and you'll never want to do that with a built-in type because such types have no `this` pointer.

Prohibiting Heap-Based Objects

Thus ends our examination of determining whether an object is on the heap. At the opposite end of the spectrum is *preventing* objects from being allocated on the heap. Here the outlook is a bit brighter. There are, as usual, three cases: objects that are directly instantiated, objects instantiated as base class parts of derived class objects, and objects embedded inside other objects. We'll consider each in turn.

Preventing clients from directly instantiating objects on the heap is easy, because such objects are always created by calls to `new` and you can make it impossible for clients to call `new`. Now, you can't affect the availability of the `new` operator (that's built into the language), but you can take advantage of the fact that the `new` operator always calls `operator new` (see Item 8), and that function is one you can declare yourself. In particular, it is one you can declare `private`. If, for example, you want to keep clients from creating `UPNumber` objects on the heap, you could do it this way:

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

Clients can now do only what they're supposed to be able to do:

```
UPNumber n1;                          // okay

static UPNumber n2;                   // also okay

UPNumber *p = new UPNumber;           // error! attempt to call
                                      // private operator new
```

It suffices to declare `operator new` private, but it looks strange to have `operator new` be private and `operator delete` be public, so unless there's a compelling reason to split up the pair, it's best to declare them in the same part of a class. If you'd like to prohibit heap-based arrays of `UPNumber` objects, too, you could declare `operator new[]` and `operator delete[]` (see Item 8) private as well. (The bond between `operator new` and `operator delete` is stronger than many people think. For information on a rarely-understood aspect of their relationship, turn to the sidebar in my article

on counting objects.)

Interestingly, declaring `operator new` private often also prevents `UPNumber` objects from being instantiated as base class parts of heap-based derived class objects. That's because `operator new` and `operator delete` are inherited, so if these functions aren't declared public in a derived class, that class inherits the private versions declared in its base(s):

```
class UPNumber { ... };              // as above

class NonNegativeUPNumber:           // assume this class
  public UPNumber {                  // declares no operator new
  ...
};

NonNegativeUPNumber n1;              // okay

static NonNegativeUPNumber n2;       // also okay

NonNegativeUPNumber *p =             // error! attempt to call
  new NonNegativeUPNumber;           // private operator new
```

If the derived class declares an `operator new` of its own, that function will be called when allocating derived class objects on the heap, and a different way will have to be found to prevent `UPNumber` base class parts from winding up there. Similarly, the fact that `UPNumber`'s `operator new` is private has no effect on attempts to allocate objects containing `UPNumber` objects as members:

```
class Asset {
public:
  Asset(int initValue);
  ...

private:
  UPNumber value;
};

Asset *pa = new Asset(100);          // fine, calls
                                     // Asset::operator new or
                                     // ::operator new, not
                                     // UPNumber::operator new
```

For all practical purposes, this brings us back to where we were when we wanted to throw an exception in the `UPNumber` constructors if a `UPNumber` object was being constructed in memory that wasn't on the heap. This time, of course, we want to throw an exception if the object in question *is* on the heap. Just as there is no portable way to determine if an address is on the heap, however, there is no portable way to determine that it is not on the heap, so we're out of luck. This should be no surprise. After all, if we could tell when an address *is* on the heap, we could surely tell when an

address is *not* on the heap. But we can't, so we can't. Oh well.

Back to Item 26: Limiting the number of objects of a class
Continue to Item 28: Smart pointers

Back to Item 27: Requiring or prohibiting heap-based objects
Continue to Item 29: Reference counting

Item 28: Smart pointers.

Smart pointers are objects that are designed to look, act, and feel like built-in pointers, but to offer greater functionality. They have a variety of applications, including resource management (see Items 9, 10, 25, and 31) and the automation of repetitive coding tasks (see Items 17 and 29).

When you use smart pointers in place of C++'s built-in pointers (i.e., *dumb* pointers), you gain control over the following aspects of pointer behavior:

- **Construction and destruction.** You determine what happens when a smart pointer is created and destroyed. It is common to give smart pointers a default value of 0 to avoid the headaches associated with uninitialized pointers. Some smart pointers are made responsible for deleting the object they point to when the last smart pointer pointing to the object is destroyed. This can go a long way toward eliminating resource leaks.
- **Copying and assignment.** You control what happens when a smart pointer is copied or is involved in an assignment. For some smart pointer types, the desired behavior is to automatically copy or make an assignment to what is pointed to, i.e., to perform a deep copy. For others, only the pointer itself should be copied or assigned. For still others, these operations should not be allowed at all. Regardless of what behavior you consider "right," the use of smart pointers lets you call the shots.
- **Dereferencing.** What should happen when a client refers to the object pointed to by a smart pointer? You get to decide. You could, for example, use smart pointers to help implement the lazy fetching strategy outlined in Item 17.

Smart pointers are generated from templates because, like built-in pointers, they must be strongly typed; the template parameter specifies the type of object pointed to. Most smart pointer templates look something like this:

```
template<class T>                    // template for smart
class SmartPtr {                     // pointer objects
public:
  SmartPtr(T* realPtr = 0);          // create a smart ptr to an
                                     // obj given a dumb ptr to
                                     // it; uninitialized ptrs
                                     // default to 0 (null)

  SmartPtr(const SmartPtr& rhs);     // copy a smart ptr

  ~SmartPtr();                       // destroy a smart ptr

  // make an assignment to a smart ptr
  SmartPtr& operator=(const SmartPtr& rhs);
```

```
  T* operator->() const;            // dereference a smart ptr
                                    // to get at a member of
                                    // what it points to

  T& operator*() const;             // dereference a smart ptr

private:
  T *pointee;                       // what the smart ptr
};                                  // points to
```

The copy constructor and assignment operator are both shown public here. For smart pointer classes where copying and assignment are not allowed, they would typically be declared private (see Item E27). The two dereferencing operators are declared `const`, because dereferencing a pointer doesn't modify it (though it may lead to modification of what the pointer points to). Finally, each smart pointer-to-`T` object is implemented by containing a dumb pointer-to-`T` within it. It is this dumb pointer that does the actual pointing.

Before going into the details of smart pointer implementation, it's worth seeing how clients might use smart pointers. Consider a distributed system in which some objects are local and some are remote. Access to local objects is generally simpler and faster than access to remote objects, because remote access may require remote procedure calls or some other way of communicating with a distant machine.

For clients writing application code, the need to handle local and remote objects differently is a nuisance. It is more convenient to have all objects appear to be located in the same place. Smart pointers allow a library to offer this illusion:

```
template<class T>                   // template for smart ptrs
class DBPtr {                       // to objects in a
public:                             // distributed DB

  DBPtr(T *realPtr = 0);            // create a smart ptr to a
                                    // DB object given a local
                                    // dumb pointer to it


  DBPtr(DataBaseID id);             // create a smart ptr to a
                                    // DB object given its
                                    // unique DB identifier

  ...                               // other smart ptr
};                                  // functions as above

class Tuple {                       // class for database
public:                             // tuples
  ...
  void displayEditDialog();         // present a graphical
                                    // dialog box allowing a
```

```
                                    // user to edit the tuple

  bool isValid() const;             // return whether *this
};                                  // passes validity check

// class template for making log entries whenever a T
// object is modified; see below for details
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};

void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);       // make log entry for this
                                    // editing operation; see
                                    // below for details

  // repeatedly display edit dialog until valid values
  // are provided
  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

The tuple to be edited inside `editTuple` may be physically located on a remote machine, but the programmer writing `editTuple` need not be concerned with such matters; the smart pointer class hides that aspect of the system. As far as the programmer is concerned, all tuples are accessed through objects that, except for how they're declared, act just like run-of-the-mill built-in pointers.

Notice the use of a `LogEntry` object in `editTuple`. A more conventional design would have been to surround the call to `displayEditDialog` with calls to begin and end the log entry. In the approach shown here, the `LogEntry`'s constructor begins the log entry and its destructor ends the log entry. As Item 9 explains, using an object to begin and end logging is more robust in the face of exceptions than explicitly calling functions, so you should accustom yourself to using classes like `LogEntry`. Besides, it's easier to create a single `LogEntry` object than to add separate calls to start and stop an entry.

As you can see, using a smart pointer isn't much different from using the dumb pointer it replaces. That's testimony to the effectiveness of encapsulation. Clients of smart pointers are *supposed* to be able to treat them as dumb pointers. As we shall see, sometimes the substitution is more transparent than others.

Construction, Assignment, and Destruction of Smart Pointers

Construction of a smart pointer is usually straightforward: locate an object to point to (typically by

using the smart pointer's constructor arguments), then make the smart pointer's internal dumb pointer point there. If no object can be located, set the internal pointer to 0 or signal an error (possibly by throwing an exception).

Implementing a smart pointer's copy constructor, assignment operator(s) and destructor is complicated somewhat by the issue of *ownership*. If a smart pointer *owns* the object it points to, it is responsible for deleting that object when it (the smart pointer) is destroyed. This assumes the object pointed to by the smart pointer is dynamically allocated. Such an assumption is common when working with smart pointers. (For ideas on how to make sure the assumption is true, see Item 27.)

Consider the `auto_ptr` template from the standard C++ library. As Item 9 explains, an `auto_ptr` object is a smart pointer that points to a heap-based object until it (the `auto_ptr`) is destroyed. When that happens, the `auto_ptr`'s destructor deletes the pointed-to object. The `auto_ptr` template might be implemented like this:

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...

private:
  T *pointee;
};
```

This works fine provided only one `auto_ptr` owns an object. But what should happen when an `auto_ptr` is copied or assigned?

```
auto_ptr<TreeNode> ptn1(new TreeNode);

auto_ptr<TreeNode> ptn2 = ptn1;      // call to copy ctor;
                                     // what should happen?

auto_ptr<TreeNode> ptn3;

ptn3 = ptn2;                         // call to operator=;
                                     // what should happen?
```

If we just copied the internal dumb pointer, we'd end up with two `auto_ptrs` pointing to the same object. This would lead to grief, because each `auto_ptr` would delete what it pointed to when the `auto_ptr` was destroyed. That would mean we'd delete an object more than once. The results of such double-deletes are undefined (and are frequently disastrous).

An alternative would be to create a new copy of what was pointed to by calling `new`. That would guarantee we didn't have too many `auto_ptrs` pointing to a single object, but it might engender an unacceptable performance hit for the creation (and later destruction) of the new object. Furthermore, we wouldn't necessarily know what type of object to create, because an `auto_ptr<T>` object need not point to an object of type `T`; it might point to an object of a type *derived* from `T`. Virtual constructors (see Item 25) can help solve this problem, but it seems inappropriate to require their use in a general-purpose class like `auto_ptr`.

The problems would vanish if `auto_ptr` prohibited copying and assignment, but a more flexible solution was adopted for the `auto_ptr` classes: object ownership is *transferred* when an `auto_ptr` is copied or assigned:

```
template<class T>
class auto_ptr {
public:
  ...

  auto_ptr(auto_ptr<T>& rhs);        // copy constructor

  auto_ptr<T>&                       // assignment
  operator=(auto_ptr<T>& rhs);       // operator

  ...
};

template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
  pointee = rhs.pointee;             // transfer ownership of
                                     // *pointee to *this

  rhs.pointee = 0;                   // rhs no longer owns
}                                    // anything

template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)                  // do nothing if this
    return *this;                    // object is being assigned
                                     // to itself

  delete pointee;                    // delete currently owned
                                     // object

  pointee = rhs.pointee;             // transfer ownership of
  rhs.pointee = 0;                   // *pointee from rhs to *this

  return *this;
}
```

Notice that the assignment operator must delete the object it owns before assuming ownership of a new object. If it failed to do this, the object would never be deleted. Remember, nobody but the `auto_ptr` object owns the object the `auto_ptr` points to.

Because object ownership is transferred when `auto_ptr`'s copy constructor is called, passing `auto_ptr`s by value is often a *very* bad idea. Here's why:

```
// this function will often lead to disaster
void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
{ s << *p; }

int main()
{
  auto_ptr<TreeNode> ptn(new TreeNode);

  ...

  printTreeNode(cout, ptn);          // pass auto_ptr by value

  ...

}
```

When `printTreeNode`'s parameter `p` is initialized (by calling `auto_ptr`'s copy constructor), ownership of the object pointed to by `ptn` is transferred to `p`. When `printTreeNode` finishes executing, `p` goes out of scope and its destructor deletes what it points to (which is what `ptn` used to point to). `ptn`, however, no longer points to anything (its underlying dumb pointer is null), so just about any attempt to use it after the call to `printTreeNode` will yield undefined behavior. Passing `auto_ptr`s by value, then, is something to be done only if you're *sure* you want to transfer ownership of an object to a (transient) function parameter. Only rarely will you want to do this.

This doesn't mean you can't pass `auto_ptr`s as parameters, it just means that pass-by-value is not the way to do it. Pass-by-reference-to-`const` is:

```
// this function behaves much more intuitively
void printTreeNode(ostream& s,
                   const auto_ptr<TreeNode>& p)
{ s << *p; }
```

In this function, `p` is a reference, not an object, so no constructor is called to initialize `p`. When `ptn` is passed to this version of `printTreeNode`, it retains ownership of the object it points to, and `ptn` can safely be used after the call to `printTreeNode`. Thus, passing `auto_ptr`s by reference-to-`const` avoids the hazards arising from pass-by-value. (For other reasons to prefer pass-by-reference to pass-by-value, check out Item E22.)

The notion of transferring ownership from one smart pointer to another during copying and assignment is interesting, but you may have been at least as interested in the unconventional declarations of the copy constructor and assignment operator. These functions normally take `const` parameters, but above they do not. In fact, the code above *changes* these parameters during the copy or the assignment. In other words, `auto_ptr` objects are modified if they are copied or are the source of an assignment!

Yes, that's exactly what's happening. Isn't it nice that C++ is flexible enough to let you do this? If the language required that copy constructors and assignment operators take `const` parameters, you'd probably have to cast away the parameters' `const`ness (see Item E21) or play other games to implement ownership transferral. Instead, you get to say exactly what you want to say: when an object is copied or is the source of an assignment, that object is changed. This may not seem intuitive, but it's simple, direct, and, in this case, accurate.

If you find this examination of `auto_ptr` member functions interesting, you may wish to see a complete implementation. You'll find one on pages 291-294, where you'll also see that the `auto_ptr` template in the standard C++ library has copy constructors and assignment operators that are more flexible than those described here. In the standard `auto_ptr` template, those functions are member function *templates*, not just member functions. (Member function templates are described later in this Item. You can also read about them in Item E25.)

A smart pointer's destructor often looks like this:

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

Sometimes there is no need for the test. An `auto_ptr` always owns what it points to, for example. At other times the test is a bit more complicated. A smart pointer that employs reference counting (see Item 29) must adjust a reference count before determining whether it has the right to delete what it points to. Of course, some smart pointers are like dumb pointers: they have no effect on the object they point to when they themselves are destroyed.

Implementing the Dereferencing Operators

Let us now turn our attention to the very heart of smart pointers, the `operator*` and `operator->` functions. The former returns the object pointed to. Conceptually, this is simple:

```
template<class T>
T& SmartPtr<T>::operator*() const
```

```
{
  perform "smart pointer" processing;

  return *pointee;
}
```

First the function does whatever processing is needed to initialize or otherwise make `pointee` valid. For example, if lazy fetching is being used (see Item 17), the function may have to conjure up a new object for `pointee` to point to. Once `pointee` is valid, the `operator*` function just returns a reference to the pointed-to object.

Note that the return type is a *reference*. It would be disastrous to return an *object* instead, though compilers will let you do it. Bear in mind that `pointee` need not point to an object of type `T`; it may point to an object of a class *derived* from `T`. If that is the case and your `operator*` function returns a `T` object instead of a reference to the actual derived class object, your function will return an object of the wrong type! (This is the *slicing problem*. See Item E22 and Item 13.) Virtual functions invoked on the object returned from your star-crossed `operator*` will not invoke the function corresponding to the dynamic type of the pointed-to object. In essence, your smart pointer will not properly support virtual functions, and how smart is a pointer like that? Besides, returning a reference is more efficient anyway, because there is no need to construct a temporary object (see Item 19). This is one of those happy occasions when correctness and efficiency go hand in hand.

If you're the kind who likes to worry, you may wonder what you should do if somebody invokes `operator*` on a null smart pointer, i.e., one whose embedded dumb pointer is null. Relax. You can do anything you want. The result of dereferencing a null pointer is undefined, so there is no "wrong" behavior. Wanna throw an exception? Go ahead, throw it. Wanna call `abort` (possibly by having an `assert` call fail)? Fine, call it. Wanna walk through memory setting every byte to your birth date modulo 256? That's okay, too. It's not nice, but as far as the language is concerned, you are completely unfettered.

The story with `operator->` is similar to that for `operator*`, but before examining `operator->`, let us remind ourselves of the unusual meaning of a call to this function. Consider again the `editTuple` function that uses a smart pointer-to-`Tuple` object:

```
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);

  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

The statement

```
pt->displayEditDialog();
```

is interpreted by compilers as:

```
(pt.operator->())->displayEditDialog();
```

That means that whatever `operator->` returns, it must be legal to apply the member-selection operator (`->`) to it. There are thus only two things `operator->` can return: a dumb pointer to an object or another smart pointer object. Most of the time, you'll want to return an ordinary dumb pointer. In those cases, you implement `operator->` as follows:

```
template<class T>
T* SmartPtr<T>::operator->() const
{
  perform "smart pointer" processing;

  return pointee;
}
```

This will work fine. Because this function returns a pointer, virtual function calls via `operator->` will behave the way they're supposed to.

For many applications, this is all you need to know about smart pointers. The reference-counting code of Item 29, for example, draws on no more functionality than we've discussed here. If you want to push your smart pointers further, however, you must know more about dumb pointer behavior and how smart pointers can and cannot emulate it. If your motto is "Most people stop at the Z — but not me!", the material that follows is for you.

Testing Smart Pointers for Nullness

With the functions we have discussed so far, we can create, destroy, copy, assign, and dereference smart pointers. One of the things we cannot do, however, is find out if a smart pointer is null:

```
SmartPtr<TreeNode> ptn;

...

if (ptn == 0) ...                    // error!

if (ptn) ...                         // error!

if (!ptn) ...                        // error!
```

This is a serious limitation.

It would be easy to add an `isNull` member function to our smart pointer classes, but that wouldn't address the problem that smart pointers don't act like dumb pointers when testing for nullness. A different approach is to provide an implicit conversion operator that allows the tests above to compile. The conversion traditionally employed for this purpose is to `void*`:

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();                 // returns 0 if the smart
  ...                               // ptr is null, nonzero
};                                  // otherwise

SmartPtr<TreeNode> ptn;

...

if (ptn == 0) ...                   // now fine

if (ptn) ...                        // also fine

if (!ptn) ...                       // fine
```

This is similar to a conversion provided by the iostream classes, and it explains why it's possible to write code like this:

```
ifstream inputFile("datafile.dat");

if (inputFile) ...                  // test to see if inputFile
                                    // was successfully
                                    // opened
```

Like all type conversion functions, this one has the drawback of letting function calls succeed that most programmers would expect to fail (see Item 5). In particular, it allows comparisons of smart pointers of completely different types:

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;

...

if (pa == po) ...                   // this compiles!
```

Even if there is no `operator==` taking a `SmartPtr<Apple>` and a `SmartPtr<Orange>`, this compiles, because both smart pointers can be implicitly converted into `void*` pointers, and there is a built-in comparison function for built-in pointers. This kind of behavior makes implicit conversion functions dangerous. (Again, see Item 5, and keep seeing it over and over until you can see it in the dark.)

There are variations on the conversion-to-`void*` motif. Some designers advocate conversion to `const void*`, others embrace conversion to `bool`. Neither of these variations eliminates the problem of allowing mixed-type comparisons.

There is a middle ground that allows you to offer a reasonable syntactic form for testing for nullness while minimizing the chances of accidentally comparing smart pointers of different types. It is to overload `operator!` for your smart pointer classes so that `operator!` returns `true` if and only if the smart pointer on which it's invoked is null:

```
template<class T>
class SmartPtr {
public:
  ...
  bool operator!() const;           // returns true if and only
  ...                               // if the smart ptr is null

};
```

This lets your clients program like this,

```
SmartPtr<TreeNode> ptn;

...

if (!ptn) {                         // fine
  ...                               // ptn is null
}
else {
  ...                               // ptn is not null
}
```

but not like this:

```
if (ptn == 0) ...                   // still an error

if (ptn) ...                        // also an error
```

The only risk for mixed-type comparisons is statements such as these:

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;

...

if (!pa == !po) ...                 // alas, this compiles
```

Fortunately, programmers don't write code like this very often. Interestingly, iostream library implementations provide an `operator!` in addition to the implicit conversion to `void*`, but these two functions typically test for slightly different stream states. (In the C++ library standard (see Item E49 and Item 35), the implicit conversion to `void*` has been replaced by an implicit conversion to `bool`, and `operator bool` always returns the negation of `operator!`.)

Converting Smart Pointers to Dumb Pointers

Sometimes you'd like to add smart pointers to an application or library that already uses dumb pointers. For example, your distributed database system may not originally have been distributed, so you may have some old library functions that aren't designed to use smart pointers:

```
class Tuple { ... };                // as before

void normalize(Tuple *pt);          // put *pt into canonical
                                    // form; note use of dumb
                                    // pointer
```

Consider what will happen if you try to call `normalize` with a smart pointer-to-`Tuple`:

```
DBPtr<Tuple> pt;

...

normalize(pt);                      // error!
```

The call will fail to compile, because there is no way to convert a `DBPtr<Tuple>` to a `Tuple*`. You can make it work by doing this,

```
normalize(&*pt);                    // gross, but legal
```

but I hope you'll agree this is repugnant.

The call can be made to succeed by adding to the smart pointer-to-T template an implicit conversion operator to a dumb pointer-to-T:

```
template<class T>                    // as before
class DBPtr {
public:
  ...
  operator T*() { return pointee; }
  ...
};

DBPtr<Tuple> pt;

...

normalize(pt);                       // this now works
```

Addition of this function also eliminates the problem of testing for nullness:

```
if (pt == 0) ...                     // fine, converts pt to a
                                     // Tuple*

if (pt) ...                          // ditto

if (!pt) ...                         // ditto (reprise)
```

However, there is a dark side to such conversion functions. (There almost always is. Have you been seeing Item 5?) They make it easy for clients to program directly with dumb pointers, thus bypassing the smarts your pointer-like objects are designed to provide:

```
void processTuple(DBPtr<Tuple>& pt)
{
  Tuple *rawTuplePtr = pt;           // converts DBPtr<Tuple> to
                                     // Tuple*

  use rawTuplePtr to modify the tuple;

}
```

Usually, the "smart" behavior provided by a smart pointer is an essential component of your design, so allowing clients to use dumb pointers typically leads to disaster. For example, if `DBPtr` implements the reference-counting strategy of Item 29, allowing clients to manipulate dumb pointers directly will almost certainly lead to bookkeeping errors that corrupt the reference-counting data structures.

Even if you provide an implicit conversion operator to go from a smart pointer to the dumb pointer

it's built on, your smart pointer will never be truly interchangeable with the dumb pointer. That's because the conversion from a smart pointer to a dumb pointer is a user-defined conversion, and compilers are forbidden from applying more than one such conversion at a time. For example, suppose you have a class representing all the clients who have accessed a particular tuple:

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);   // pt identifies the
  ...                                // tuple whose accessors
};                                   // we care about
```

As usual, `TupleAccessors`' single-argument constructor also acts as a type-conversion operator from `Tuple*` to `TupleAccessors` (see Item 5). Now consider a function for merging the information in two `TupleAccessors` objects:

```
TupleAccessors merge(const TupleAccessors& ta1,
                     const TupleAccessors& ta2);
```

Because a `Tuple*` may be implicitly converted to a `TupleAccessors`, calling `merge` with two dumb `Tuple*` pointers is fine:

```
Tuple *pt1, *pt2;

...

merge(pt1, pt2);                     // fine, both pointers are converted
                                     // to TupleAccessors objects
```

The corresponding call with smart `DBPtr<Tuple>` pointers, however, fails to compile:

```
DBPtr<Tuple> pt1, pt2;

...

merge(pt1, pt2);                     // error! No way to convert pt1 and
                                     // pt2 to TupleAccessors objects
```

That's because a conversion from `DBPtr<Tuple>` to `TupleAccessors` calls for *two* user-defined conversions (one from `DBPtr<Tuple>` to `Tuple*` and one from `Tuple*` to `TupleAccessors`), and such sequences of conversions are prohibited by the language.

Smart pointer classes that provide an implicit conversion to a dumb pointer open the door to a

particularly nasty bug. Consider this code:

```
DBPtr<Tuple> pt = new Tuple;

...

delete pt;
```

This should not compile. After all, `pt` is not a pointer, it's an object, and you can't delete an object. Only pointers can be deleted, right?

Right. But remember from Item 5 that compilers use implicit type conversions to make function calls succeed whenever they can, and recall from Item 8 that use of the `delete` operator leads to calls to a destructor and to `operator delete`, both of which are functions. Compilers want these function calls to succeed, so in the `delete` statement above, they implicitly convert `pt` to a `Tuple*`, then they delete that. This will almost certainly break your program.

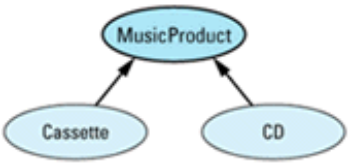
If `pt` owns the object it points to, that object is now deleted twice, once at the point where `delete` is called, a second time when `pt`'s destructor is invoked. If `pt` doesn't own the object, somebody else does. That somebody may be the person who deleted `pt`, in which case all is well. If, however, the owner of the object pointed to by `pt` is not the person who deleted `pt`, we can expect the rightful owner to delete that object again later. The first and last of these scenarios leads to an object being deleted twice, and deleting an object more than once yields undefined behavior.

This bug is especially pernicious because the whole idea behind smart pointers is to make them look and feel as much like dumb pointers as possible. The closer you get to this ideal, the more likely your clients are to forget they are using smart pointers. If they do, who can blame them if they continue to think that in order to avoid resource leaks, they must call `delete` if they called `new`?

The bottom line is simple: don't provide implicit conversion operators to dumb pointers unless there is a compelling reason to do so.

Smart Pointers and Inheritance-Based Type Conversions

Suppose we have a public inheritance hierarchy modeling consumer products for storing music:




```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};

class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};

class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

Further suppose we have a function that, given a `MusicProduct` object, displays the title of the product and then plays it:

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

Such a function might be used like this:

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");

displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

There are no surprises here, but look what happens if we replace the dumb pointers with their allegedly smart counterparts:

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);         // error!
displayAndPlay(nightmareMusic, 0);    // error!
```

If smart pointers are so brainy, why won't these compile?

They won't compile because there is no conversion from a `SmartPtr<CD>` or a `SmartPtr<Cassette>` to a `SmartPtr<MusicProduct>`. As far as compilers are concerned, these are three separate classes — they have no relationship to one another. Why should compilers think otherwise? After all, it's not like `SmartPtr<CD>` or `SmartPtr<Cassette>` inherits from `SmartPtr<MusicProduct>`. With no inheritance relationship between these classes, we can hardly expect compilers to run around converting objects of one type to objects of other types.

Fortunately, there is a way to get around this limitation, and the idea (if not the practice) is simple: give each smart pointer class an implicit type conversion operator (see Item 5) for each smart pointer class to which it should be implicitly convertible. For example, in the music hierarchy, you'd add an `operator SmartPtr<MusicProduct>` to the smart pointer classes for `Cassette` and `CD`:

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  Cassette *pointee;
};

class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
```

```
  CD *pointee;
};
```

The drawbacks to this approach are twofold. First, you must manually specialize the `SmartPtr` class instantiations so you can add the necessary implicit type conversion operators, but that pretty much defeats the purpose of templates. Second, you may have to add many such conversion operators, because your pointed-to object may be deep in an inheritance hierarchy, and you must provide a conversion operator for *each* base class from which that object directly or indirectly inherits. (If you think you can get around this by providing only an implicit type conversion operator for each direct base class, think again. Because compilers are prohibited from employing more than one user-defined type conversion function at a time, they can't convert a smart pointer-to-`T` to a smart pointer-to-indirect-base-class-of-`T` unless they can do it in a single step.)

It would be quite the time-saver if you could somehow get compilers to write all these implicit type conversion functions for you. Thanks to a recent language extension, you can. The extension in question is the ability to declare (nonvirtual) *member function templates* (usually just called *member templates*), and you use it to generate smart pointer conversion functions like this:

```
template<class T>                    // template class for smart
class SmartPtr {                     // pointers-to-T objects
public:
  SmartPtr(T* realPtr = 0);

  T* operator->() const;
  T& operator*() const;

  template<class newType>             // template function for
  operator SmartPtr<newType>()        // implicit conversion ops.
  {
    return SmartPtr<newType>(pointee);
  }

  ...
};
```

Now hold on to your headlights, this isn't magic — but it's close. It works as follows. (I'll give a specific example in a moment, so don't despair if the remainder of this paragraph reads like so much gobbledygook. After you've seen the example, it'll make more sense, I promise.) Suppose a compiler has a smart pointer-to-`T` object, and it's faced with the need to convert that object into a smart pointer-to-base-class-of-`T`. The compiler checks the class definition for `SmartPtr<T>` to see if the requisite conversion operator is declared, but it is not. (It can't be: no conversion operators are declared in the template above.) The compiler then checks to see if there's a member function template it can instantiate that would let it perform the conversion it's looking for. It finds such a template (the one taking the formal type parameter `newType`), so it instantiates the template with `newType` bound to the base class of `T` that's the target of the conversion. At that point, the only

question is whether the code for the instantiated member function will compile. In order for it to compile, it must be legal to pass the (dumb) pointer `pointee` to the constructor for the smart pointer-to-base-of-`T`. `pointee` is of type `T`, so it is certainly legal to convert it into a pointer to its (public or protected) base classes. Hence, the code for the type conversion operator will compile, and the implicit conversion from smart pointer-to-`T` to smart pointer-to-base-of-`T` will succeed.

An example will help. Let us return to the music hierarchy of CDs, cassettes, and music products. We saw earlier that the following code wouldn't compile, because there was no way for compilers to convert the smart pointers to CDs or cassettes into smart pointers to music products:

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);         // used to be an error
displayAndPlay(nightmareMusic, 0);    // used to be an error
```

With the revised smart pointer class containing the member function template for implicit type conversion operators, this code will succeed. To see why, look at this call:

```
displayAndPlay(funMusic, 10);
```

The object `funMusic` is of type `SmartPtr<Cassette>`. The function `displayAndPlay` expects a `SmartPtr<MusicProduct>` object. Compilers detect the type mismatch and seek a way to convert `funMusic` into a `SmartPtr<MusicProduct>` object. They look for a single-argument constructor (see Item 5) in the `SmartPtr<MusicProduct>` class that takes a `SmartPtr<Cassette>`, but they find none. They look for an implicit type conversion operator in the `SmartPtr<Cassette>` class that yields a `SmartPtr<MusicProduct>` class, but that search also fails. They then look for a member function template they can instantiate to yield one of these functions. They discover that the template inside `SmartPtr<Cassette>`, when instantiated with `newType` bound to `MusicProduct`, generates the necessary function. They instantiate the function, yielding the following code:

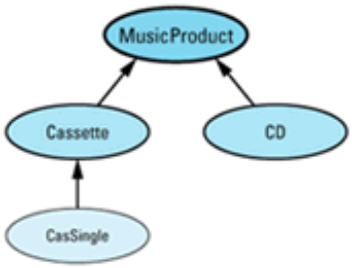
```
SmartPtr<Cassette>::  operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
}
```

Will this compile? For all intents and purposes, nothing is happening here except the calling of the `SmartPtr<MusicProduct>` constructor with `pointee` as its argument, so the real question is whether one can construct a `SmartPtr<MusicProduct>` object with a `Cassette*` pointer. The

`SmartPtr<MusicProduct>` constructor expects a `MusicProduct*` pointer, but now we're on the familiar ground of conversions between dumb pointer types, and it's clear that `Cassette*` can be passed in where a `MusicProduct*` is expected. The construction of the `SmartPtr<MusicProduct>` is therefore successful, and the conversion of the `SmartPtr<Cassette>` to `SmartPtr<MusicProduct>` is equally successful. *Voilà!* Implicit conversion of smart pointer types. What could be simpler?

Furthermore, what could be more powerful? Don't be misled by this example into assuming that this works only for pointer conversions up an inheritance hierarchy. The method shown succeeds for *any* legal implicit conversion between pointer types. If you've got a dumb pointer type `T1*` and another dumb pointer type `T2*`, you can implicitly convert a smart pointer-to-`T1` to a smart pointer-to-`T2` if and only if you can implicitly convert a `T1*` to a `T2*`.

This technique gives you exactly the behavior you want — almost. Suppose we augment our `MusicProduct` hierarchy with a new class, `CasSingle`, for representing cassette singles. The revised hierarchy looks like this:



Now consider this code:

```
template<class T>                    // as above, including member tem-
class SmartPtr { ... };              // plate for conversion operators

void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);

SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));

displayAndPlay(dumbMusic, 1);        // error!
```

In this example, `displayAndPlay` is overloaded, with one function taking a

`SmartPtr<MusicProduct>` object and the other taking a `SmartPtr<Cassette>` object. When we invoke `displayAndPlay` with a `SmartPtr<CasSingle>`, we expect the `SmartPtr<Cassette>` function to be chosen, because `CasSingle` inherits directly from `Cassette` and only indirectly from `MusicProduct`. Certainly that's how it would work with dumb pointers. Alas, our smart pointers aren't that smart. They employ member functions as conversion operators, and as far as C++ compilers are concerned, all calls to conversion functions are equally good. As a result, the call to `displayAndPlay` is ambiguous, because the conversion from `SmartPtr<CasSingle>` to `SmartPtr<Cassette>` is no better than the conversion to `SmartPtr<MusicProduct>`.

Implementing smart pointer conversions through member templates has two additional drawbacks. First, support for member templates is rare, so this technique is currently anything but portable. In the future, that will change, but nobody knows just how far in the future that will be. Second, the mechanics of why this works are far from transparent, relying as they do on a detailed understanding of argument-matching rules for function calls, implicit type conversion functions, implicit instantiation of template functions, and the existence of member function templates. Pity the poor programmer who has never seen this trick before and is then asked to maintain or enhance code that relies on it. The technique is clever, that's for sure, but too much cleverness can be a dangerous thing.

Let's stop beating around the bush. What we really want to know is how we can make smart pointer classes behave just like dumb pointers for purposes of inheritance-based type conversions. The answer is simple: we can't. As Daniel Edelson has noted, smart pointers are smart, but they're not pointers. The best we can do is to use member templates to generate conversion functions, then use casts (see Item 2) in those cases where ambiguity results. This isn't a perfect state of affairs, but it's pretty good, and having to cast away ambiguity in a few cases is a small price to pay for the sophisticated functionality smart pointers can provide.

Smart Pointers and const

Recall that for dumb pointers, `const` can refer to the thing pointed to, to the pointer itself, or both (see Item E21):

```
CD goodCD("Flood");

const CD *p;                          // p is a non-const pointer
                                      // to a const CD object

CD * const p = &goodCD;               // p is a const pointer to
                                      // a non-const CD object;
                                      // because p is const, it
                                      // must be initialized

const CD * const p = &goodCD;         // p is a const pointer to
                                      // a const CD object
```

Naturally, we'd like to have the same flexibility with smart pointers. Unfortunately, there's only one place to put the `const`, and there it applies to the pointer, not to the object pointed to:

```
const SmartPtr<CD> p =                    // p is a const smart ptr
  &goodCD;                                // to a non-const CD object
```

This seems simple enough to remedy — just create a smart pointer to a *`const`* `CD`:

```
SmartPtr<const CD> p =                    // p is a non-const smart ptr
  &goodCD;                                // to a const CD object
```

Now we can create the four combinations of `const` and non-`const` objects and pointers we seek:

```
SmartPtr<CD> p;                           // non-const object,
                                          // non-const pointer

SmartPtr<const CD> p;                     // const object,
                                          // non-const pointer

const SmartPtr<CD> p = &goodCD;           // non-const object,
                                          // const pointer

const SmartPtr<const CD> p = &goodCD;     // const object,
                                          // const pointer
```

Alas, this ointment has a fly in it. Using dumb pointers, we can initialize `const` pointers with non-`const` pointers and we can initialize pointers to `const` objects with pointers to non-`const`s; the rules for assignments are analogous. For example:

```
CD *pCD = new CD("Famous Movie Themes");

const CD * pConstCD = pCD;                // fine
```

But look what happens if we try the same thing with smart pointers:

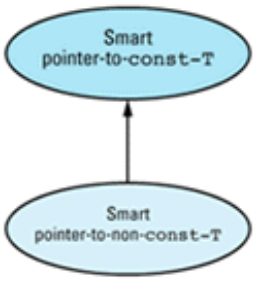
```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtr<const CD> pConstCD = pCD;        // fine?
```

`SmartPtr<CD>` and `SmartPtr<const CD>` are completely different types. As far as your compilers know, they are unrelated, so they have no reason to believe they are assignment-compatible. In

what must be an old story by now, the only way these two types will be considered assignment-compatible is if you've provided a function to convert objects of type `SmartPtr<CD>` to objects of type `SmartPtr<const CD>`. If you've got a compiler that supports member templates, you can use the technique shown above for automatically generating the implicit type conversion operators you need. (I remarked earlier that the technique worked anytime the corresponding conversion for dumb pointers would work, and I wasn't kidding. Conversions involving `const` are no exception.) If you don't have such a compiler, you have to jump through one more hoop.

Conversions involving `const` are a one-way street: it's safe to go from non-`const` to `const`, but it's not safe to go from `const` to non-`const`. Furthermore, anything you can do with a `const` pointer you can do with a non-`const` pointer, but with non-`const` pointers you can do other things, too (for example, assignment). Similarly, anything you can do with a pointer-to-`const` is legal for a pointer-to-non-`const`, but you can do some things (such as assignment) with pointers-to-non-`const`s that you can't do with pointers-to-`const`s.

These rules sound like the rules for public inheritance (see Item E35). You can convert from a derived class object to a base class object, but not vice versa, and you can do anything to a derived class object you can do to a base class object, but you can typically do additional things to a derived class object, as well. We can take advantage of this similarity when implementing smart pointers by having each smart pointer-to-`T` class publicly inherit from a corresponding smart pointer-to-`const-T` class:




```
template<class T>                    // smart pointers to const
class SmartPtrToConst {              // objects

  ...                                // the usual smart pointer
                                     // member functions

protected:
  union {
    const T* constPointee;           // for SmartPtrToConst access
    T* pointee;                      // for SmartPtr access
  };
};
```

```
template<class T>                    // smart pointers to
class SmartPtr:                      // non-const objects
  public SmartPtrToConst<T> {
  ...                                // no data members
};
```

With this design, the smart pointer-to-non-`const-T` object needs to contain a dumb pointer-to-non-`const-T`, and the smart pointer-to-`const-T` needs to contain a dumb pointer-to-`const-T`. The naive way to handle this would be to put a dumb pointer-to-`const-T` in the base class and a dumb pointer-to-non-`const-T` in the derived class. That would be wasteful, however, because `SmartPtr` objects would contain two dumb pointers: the one they inherited from `SmartPtrToConst` and the one in `SmartPtr` itself.

This problem is resolved by employing that old battle axe of the C world, a union, which can be as useful in C++ as it is in C. The union is protected, so both classes have access to it, and it contains both of the necessary dumb pointer types. `SmartPtrToConst<T>` objects use the `constPointee` pointer, `SmartPtr<T>` objects use the `pointee` pointer. We therefore get the advantages of two different pointers without having to allocate space for more than one. (See Item E10 for another example of this.) Such is the beauty of a union. Of course, the member functions of the two classes must constrain themselves to using only the appropriate pointer, and you'll get no help from compilers in enforcing that constraint. Such is the risk of a union.

With this new design, we get the behavior we want:

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtrToConst<CD> pConstCD = pCD;     // fine
```

Evaluation

That wraps up the subject of smart pointers, but before we leave the topic, we should ask this question: are they worth the trouble, especially if your compilers lack support for member function templates?

Often they are. The reference-counting code of Item 29, for example, is greatly simplified by using smart pointers. Furthermore, as that example demonstrates, some uses of smart pointers are sufficiently limited in scope that things like testing for nullness, conversion to dumb pointers, inheritance-based conversions, and support for pointers-to-`consts` are irrelevant. At the same time, smart pointers can be tricky to implement, understand, and maintain. Debugging code using smart pointers is more difficult than debugging code using dumb pointers. Try as you may, you will never succeed in designing a general-purpose smart pointer that can seamlessly replace its dumb pointer counterpart.

Smart pointers nevertheless make it possible to achieve effects in your code that would otherwise be difficult to implement. Smart pointers should be used judiciously, but every C++ programmer will find them useful at one time or another.

Back to Item 27: Requiring or prohibiting heap-based objects
Continue to Item 29: Reference counting

Back to Item 28: Smart pointers
Continue to Item 30: Proxy classes

Item 29: Reference counting.

Reference counting is a technique that allows multiple objects with the same value to share a single representation of that value. There are two common motivations for the technique. The first is to simplify the bookkeeping surrounding heap objects. Once an object is allocated by calling `new`, it's crucial to keep track of who *owns* that object, because the owner — and only the owner — is responsible for calling `delete` on it. But ownership can be transferred from object to object as a program runs (by passing pointers as parameters, for example), so keeping track of an object's ownership is hard work. Classes like `auto_ptr` (see Item 9) can help with this task, but experience has shown that most programs still fail to get it right. Reference counting eliminates the burden of tracking object ownership, because when an object employs reference counting, it owns itself. When nobody is using it any longer, it destroys itself automatically. Thus, reference counting constitutes a simple form of garbage collection.

The second motivation for reference counting is simple common sense. If many objects have the same value, it's silly to store that value more than once. Instead, it's better to let all the objects with that value share its representation. Doing so not only saves memory, it also leads to faster-running programs, because there's no need to construct and destruct redundant copies of the same object value.

Like most simple ideas, this one hovers above a sea of interesting details. God may or may not be in the details, but successful implementations of reference counting certainly are. Before delving into details, however, let us master basics. A good way to begin is by seeing how we might come to have many objects with the same value in the first place. Here's one way:

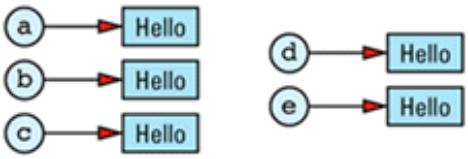
```
class String {                      // the standard string type may
public:                             // employ the techniques in this
                                    // Item, but that is not required

  String(const char *value = "");
  String& operator=(const String& rhs);
  ...

private:
  char *data;
};

String a, b, c, d, e;

a = b = c = d = e = "Hello";
```

It should be apparent that objects `a` through `e` all have the same value, namely `"Hello"`. How that value is represented depends on how the `String` class is implemented, but a common

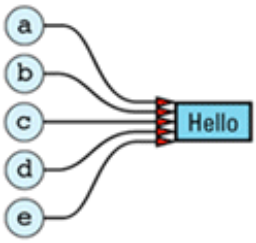
implementation would have each `String` object carry its own copy of the value. For example, `String`'s assignment operator might be implemented like this:

```
String& String::operator=(const String& rhs)
{
  if (this == &rhs) return *this;         // see Item E17

  delete [] data;
  data =   new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;                           // see Item E15
}
```

Given this implementation, we can envision the five objects and their values as follows:



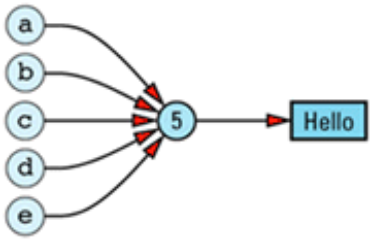
The redundancy in this approach is clear. In an ideal world, we'd like to change the picture to look like this:



Here only one copy of the value `"Hello"` is stored, and all the `String` objects with that value share its representation.

In practice, it isn't possible to achieve this ideal, because we need to keep track of how many objects are sharing a value. If object `a` above is assigned a different value from `"Hello"`, we can't destroy the value `"Hello"`, because four other objects still need it. On the other hand, if only a single object had the value `"Hello"` and that object went out of scope, no object would have that value and we'd have to destroy the value to avoid a resource leak.

The need to store information on the number of objects currently sharing — *referring to* — a value means our ideal picture must be modified somewhat to take into account the existence of a *reference count*:



(Some people call this number a *use count*, but I am not one of them. C++ has enough idiosyncrasies of its own; the last thing it needs is terminological factionalism.)

Implementing Reference Counting

Creating a reference-counted `String` class isn't difficult, but it does require attention to detail, so we'll walk through the implementation of the most common member functions of such a class. Before we do that, however, it's important to recognize that we need a place to store the reference count for each `String` value. That place cannot be in a `String` object, because we need one reference count per string *value*, not one reference count per string *object*. That implies a coupling between values and reference counts, so we'll create a class to store reference counts and the values they track. We'll call this class `StringValue`, and because its only *raison d'être* is to help implement the `String` class, we'll nest it inside `String`'s private section. Furthermore, it will be convenient to give all the member functions of `String` full access to the `StringValue` data structure, so we'll declare `StringValue` to be a `struct`. This is a trick worth knowing: nesting a struct in the private part of a class is a convenient way to give access to the struct to all the members of the class, but to deny access to everybody else (except, of course, friends of the class).

Our basic design looks like this:

```
class String {
public:
  ...                                  // the usual String member
                                       // functions go here

private:
  struct StringValue { ... };          // holds a reference count
                                       // and a string value

  StringValue *value;                  // value of this String
};
```

We could give this class a different name (`RCString`, perhaps) to emphasize that it's implemented using reference counting, but the implementation of a class shouldn't be of concern to clients of that class. Rather, clients should interest themselves only in a class's public interface. Our reference-counting implementation of the `String` interface supports exactly the same operations as a non-reference-counted version, so why muddy the conceptual waters by embedding implementation decisions in the names of classes that correspond to abstract concepts? Why indeed? So we don't.

Here's `StringValue`:

```
class String {
private:
  struct StringValue {
    int refCount;
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

  ...

};

String::StringValue::StringValue(const char *initValue)
: refCount(1)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}



String::StringValue::~StringValue()
{
  delete [] data;
}
```

That's all there is to it, and it should be clear that's nowhere near enough to implement the full functionality of a reference-counted string. For one thing, there's neither a copy constructor nor an assignment operator (see Item E11), and for another, there's no manipulation of the `refCount` field. Worry not — the missing functionality will be provided by the `String` class. The primary purpose of `StringValue` is to give us a place to associate a particular value with a count of the number of `String` objects sharing that value. `StringValue` gives us that, and that's enough.

We're now ready to walk our way through `String`'s member functions. We'll begin with the constructors:

```
class String {
public:
```

```
  String(const char *initValue = "");
  String(const String& rhs);


  ...


};
```

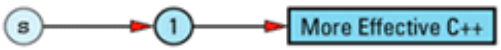
The first constructor is implemented about as simply as possible. We use the passed-in `char*` string to create a new `StringValue` object, then we make the `String` object we're constructing point to the newly-minted `StringValue`:

```
String::String(const char *initValue)
: value(new StringValue(initValue))
{}
```

For client code that looks like this,

```
String s("More Effective C++");
```

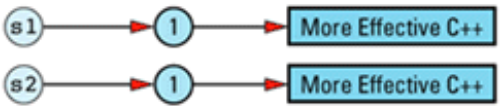
we end up with a data structure that looks like this:



`String` objects constructed separately, but with the same initial value do not share a data structure, so client code of this form,

```
String s1("More Effective C++");
String s2("More Effective C++");
```

yields this data structure:



It is possible to eliminate such duplication by having `String` (or `StringValue`) keep track of existing `StringValue` objects and create new ones only for truly unique strings, but such refinements on reference counting are somewhat off the beaten path. As a result, I'll leave them in the form of the feared and hated exercise for the reader.

The `String` copy constructor is not only unfeared and unhated, it's also efficient: the newly created `String` object shares the same `StringValue` object as the `String` object that's being copied:

```
String::String(const String& rhs)
: value(rhs.value)
{
  ++value->refCount;
}
```

Graphically, code like this,

```
String s1("More Effective C++");
String s2 = s1;
```

results in this data structure:

This is substantially more efficient than a conventional (non-reference-counted) `String` class, because there is no need to allocate memory for the second copy of the string value, no need to deallocate that memory later, and no need to copy the value that would go in that memory. Instead, we merely copy a pointer and increment a reference count.

The `String` destructor is also easy to implement, because most of the time it doesn't do anything. As long as the reference count for a `StringValue` is non-zero, at least one `String` object is using the value; it must therefore not be destroyed. Only when the `String` being destructed is the sole user of the value — i.e., when the value's reference count is 1 — should the `String` destructor destroy the `StringValue` object:

```
class String {
public:
  ~String();
  ...
};


String::~String()
{
  if (--value->refCount == 0) delete value;
}
```

Compare the efficiency of this function with that of the destructor for a non-reference-counted

implementation. Such a function would always call `delete` and would almost certainly have a nontrivial runtime cost. Provided that different `String` objects do in fact sometimes have the same values, the implementation above will sometimes do nothing more than decrement a counter and compare it to zero.

If, at this point, the appeal of reference counting is not becoming apparent, you're just not paying attention.

That's all there is to `String` construction and destruction, so we'll move on to consideration of the `String` assignment operator:

```
class String {
public:
  String& operator=(const String& rhs);
  ...


};
```

When a client writes code like this,

```
s1 = s2;                              // s1 and s2 are both String objects
```

the result of the assignment should be that `s1` and `s2` both point to the same `StringValue` object. That object's reference count should therefore be incremented during the assignment. Furthermore, the `StringValue` object that `s1` pointed to prior to the assignment should have its reference count decremented, because `s1` will no longer have that value. If `s1` was the only `String` with that value, the value should be destroyed. In C++, all that looks like this:

```
String& String::operator=(const String& rhs)
{
  if (value == rhs.value) {          // do nothing if the values
    return *this;                    // are already the same; this
  }                                  // subsumes the usual test of
                                     // this against &rhs (see Item E17)

  if (--value->refCount == 0) {      // destroy *this's value if
    delete value;                    // no one else is using it
  }

  value = rhs.value;                 // have *this share rhs's
  ++value->refCount;                 // value

  return *this;
}
```

Copy-on-Write

To round out our examination of reference-counted strings, consider an array-bracket operator (`[]`), which allows individual characters within strings to be read and written:

```
class String {
public:
  const char&
    operator[](int index) const;       // for const Strings

  char& operator[](int index);         // for non-const Strings

...

};
```

Implementation of the `const` version of this function is straightforward, because it's a read-only operation; the value of the string can't be affected:

```
const char& String::operator[](int index) const
{
  return value->data[index];
}
```

(This function performs sanity checking on `index` in the grand C++ tradition, which is to say not at all. As usual, if you'd like a greater degree of parameter validation, it's easy to add.)

The non-`const` version of `operator[]` is a completely different story. This function may be called to read a character, but it might be called to write one, too:

```
String s;

...

cout << s[3];                        // this is a read
s[5] = 'x';                          // this is a write
```

We'd like to deal with reads and writes differently. A simple read can be dealt with in the same way as the `const` version of `operator[]` above, but a write must be implemented in quite a different fashion.

When we modify a `String`'s value, we have to be careful to avoid modifying the value of other `String` objects that happen to be sharing the same `StringValue` object. Unfortunately, there is no way for C++ compilers to tell us whether a particular use of `operator[]` is for a read or a write, so we must be pessimistic and assume that *all* calls to the non-`const` `operator[]` are for writes. (Proxy classes can help us differentiate reads from writes — see Item 30.)

To implement the non-`const` `operator[]` safely, we must ensure that no other `String` object shares

the `StringValue` to be modified by the presumed write. In short, we must ensure that the reference count for a `String`'s `StringValue` object is exactly one any time we return a reference to a character inside that `StringValue` object. Here's how we do it:

```
char& String::operator[](int index)
{
  // if we're sharing a value with other String objects,
  // break off a separate copy of the value for ourselves
  if (value->refCount > 1) {
    --value->refCount;                    // decrement current value's
                                          // refCount, because we won't
                                          // be using that value any more

    value =                               // make a copy of the
      new StringValue(value->data);       // value for ourselves
  }

  // return a reference to a character inside our
  // unshared StringValue object
  return value->data[index];
}
```

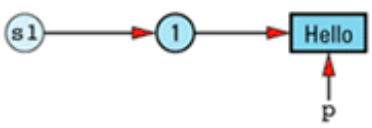
This idea — that of sharing a value with other objects until we have to write on our own copy of the value — has a long and distinguished history in Computer Science, especially in operating systems, where processes are routinely allowed to share pages until they want to modify data on their own copy of a page. The technique is common enough to have a name: *copy-on-write*. It's a specific example of a more general approach to efficiency, that of lazy evaluation (see Item 17).

Pointers, References, and Copy-on-Write

This implementation of copy-on-write allows us to preserve both efficiency and correctness — almost. There is one lingering problem. Consider this code:

```
String s1 = "Hello";


char *p = &s1[1];
```

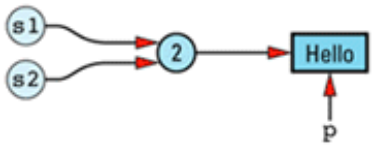
Our data structure at this point looks like this:



Now consider an additional statement:

```
String s2 = s1;
```

The `String` copy constructor will make `s2` share `s1`'s `StringValue`, so the resulting data structure will be this one:



The implications of a statement such as the following, then, are not pleasant to contemplate:

```
*p = 'x';                       // modifies both s1 and s2!
```

There is no way the `String` copy constructor can detect this problem, because it has no way to know that a pointer into `s1`'s `StringValue` object exists. And this problem isn't limited to pointers: it would exist if someone had saved a *reference* to the result of a call to `String`'s non-`const` `operator[]`.

There are at least three ways of dealing with this problem. The first is to ignore it, to pretend it doesn't exist. This approach turns out to be distressingly common in class libraries that implement reference-counted strings. If you have access to a reference-counted string, try the above example and see if you're distressed, too. If you're not sure if you have access to a reference-counted string, try the example anyway. Through the wonder of encapsulation, you may be using such a type without knowing it.

Not all implementations ignore such problems. A slightly more sophisticated way of dealing with such difficulties is to define them out of existence. Implementations adopting this strategy typically put something in their documentation that says, more or less, "Don't do that. If you do, results are undefined." If you then do it anyway — wittingly or no — and complain about the results, they respond, "Well, we *told* you not to do that." Such implementations are often efficient, but they leave much to be desired in the usability department.

There is a third solution, and that's to eliminate the problem. It's not difficult to implement, but it can reduce the amount of value sharing between objects. Its essence is this: add a flag to each `StringValue` object indicating whether that object is shareable. Turn the flag on initially (the object is shareable), but turn it off whenever the non-`const` `operator[]` is invoked on the value represented by that object. Once the flag is set to `false`, it stays that way forever.[10]

Here's a modified version of `StringValue` that includes a shareability flag:

```
class String {
private:
```

```
  struct StringValue {
    int refCount;
    bool shareable;               // add this
    char *data;


    StringValue(const char *initValue);
    ~StringValue();


  };


...


};


String::StringValue::StringValue(const char *initValue)
:   refCount(1),
    shareable(true)               // add this
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}


String::StringValue::~StringValue()
{
  delete [] data;
}
```

As you can see, not much needs to change; the two lines that require modification are flagged with comments. Of course, `String`'s member functions must be updated to take the `shareable` field into account. Here's how the copy constructor would do that:

```
String::String(const String& rhs)
{
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }


  else {
    value = new StringValue(rhs.value->data);
  }
```

```
}
```

All the other `String` member functions would have to check the `shareable` field in an analogous fashion. The non-`const` version of `operator[]` would be the only function to set the `shareable` flag to `false`:

```
char& String::operator[](int index)
{
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }


  value->shareable = false;           // add this


  return value->data[index];
}
```

If you use the proxy class technique of Item 30 to distinguish read usage from write usage in `operator[]`, you can usually reduce the number of `StringValue` objects that must be marked unshareable.

A Reference-Counting Base Class

Reference counting is useful for more than just strings. Any class in which different objects may have values in common is a legitimate candidate for reference counting. Rewriting a class to take advantage of reference counting can be a lot of work, however, and most of us already have more than enough to do. Wouldn't it be nice if we could somehow write (and test and document) the reference counting code in a context-independent manner, then just graft it onto classes when needed? Of course it would. In a curious twist of fate, there's a way to do it (or at least to do most of it).

The first step is to create a base class, `RCObject`, for reference-counted objects. Any class wishing to take advantage of automatic reference counting must inherit from this class. `RCObject` encapsulates the reference count itself, as well as functions for incrementing and decrementing that count. It also contains the code for destroying a value when it is no longer in use, i.e., when its reference count becomes 0. Finally, it contains a field that keeps track of whether this value is shareable, and it provides functions to query this value and set it to false. There is no need for a function to set the shareability field to true, because all values are shareable by default. As noted above, once an object has been tagged unshareable, there is no way to make it shareable again.

`RCObject`'s class definition looks like this:

```
class RCObject {
```

```
public:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;


  void addReference();
  void removeReference();


  void markUnshareable();
  bool isShareable() const;


  bool isShared() const;


private:
  int refCount;
  bool shareable;
};
```

`RCObject`s can be created (as the base class parts of more derived objects) and destroyed; they can have new references added to them and can have current references removed; their shareability status can be queried and can be disabled; and they can report whether they are currently being shared. That's all they offer. As a class encapsulating the notion of being reference-countable, that's really all we have a right to expect them to do. Note the tell-tale virtual destructor, a sure sign this class is designed for use as a base class (see Item E14). Note also how the destructor is a *pure* virtual function, a sure sign this class is designed to be used *only* as a base class.

The code to implement `RCObject` is, if nothing else, brief:

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}              // virtual dtors must always
                                      // be implemented, even if
                                      // they are pure virtual
                                      // and do nothing (see also
                                      // Item 33 and Item E14)

void RCObject::addReference() { ++refCount; }
```

```
void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

Curiously, we set `refCount` to 0 inside both constructors. This seems counterintuitive. Surely at least the creator of the new `RCObject` is referring to it! As it turns out, it simplifies things for the creators of `RCObjects` to set `refCount` to 1 themselves, so we oblige them here by not getting in their way. We'll get a chance to see the resulting code simplification shortly.

Another curious thing is that the copy constructor always sets `refCount` to 0, regardless of the value of `refCount` for the `RCObject` we're copying. That's because we're creating a new object representing a value, and new values are always unshared and referenced only by their creator. Again, the creator is responsible for setting the `refCount` to its proper value.

The `RCObject` assignment operator looks downright subversive: it does *nothing*. Frankly, it's unlikely this operator will ever be called. `RCObject` is a base class for a shared *value* object, and in a system based on reference counting, such objects are not assigned to one another, objects *pointing* to them are. In our case, we don't expect `StringValue` objects to be assigned to one another, we expect only `String` objects to be involved in assignments. In such assignments, no change is made to the value of a `StringValue` — only the `StringValue` reference count is modified.

Nevertheless, it is conceivable that some as-yet-unwritten class might someday inherit from `RCObject` and might wish to allow assignment of reference-counted values (see Item 32 and Item E16). If so, `RCObject`'s assignment operator should do the right thing, and the right thing is to do nothing. To see why, imagine that we wished to allow assignments between `StringValue` objects. Given `StringValue` objects `sv1` and `sv2`, what should happen to `sv1`'s and `sv2`'s reference counts in an assignment?

```
sv1 = sv2;                      // how are sv1's and sv2's reference
                                // counts affected?
```

Before the assignment, some number of `String` objects are pointing to `sv1`. That number is unchanged by the assignment, because only `sv1`'s *value* changes. Similarly, some number of `String` objects are pointing to `sv2` prior to the assignment, and after the assignment, exactly the same `String` objects point to `sv2`. `sv2`'s reference count is also unchanged. When `RCObjects` are involved in an assignment, then, the number of objects pointing to those objects is unaffected, hence `RCObject::operator=` should change no reference counts. That's exactly what the implementation above does. Counterintuitive? Perhaps, but it's still correct.

The code for `RCObject::removeReference` is responsible not only for decrementing the object's `refCount`, but also for destroying the object if the new value of `refCount` is 0. It accomplishes this latter task by `delete`ing `this`, which, as Item 27 explains, is safe only if we know that `*this` is a heap object. For this class to be successful, we must engineer things so that `RCObjects` can be created only on the heap. General approaches to achieving that end are discussed in Item 27, but the specific measures we'll employ in this case are described at the conclusion of this Item.

To take advantage of our new reference-counting base class, we modify `StringValue` to inherit its reference counting capabilities from `RCObject`:

```
class String {
private:
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    ~StringValue();

  };

...

};

String::StringValue::StringValue(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

This version of `StringValue` is almost identical to the one we saw earlier. The only thing that's changed is that `StringValue`'s member functions no longer manipulate the `refCount` field. `RCObject` now handles what they used to do.

Don't feel bad if you blanched at the sight of a nested class (`StringValue`) inheriting from a class (`RCObject`) that's unrelated to the nesting class (`String`). It looks weird to everybody at first, but it's perfectly kosher. A nested class is just as much a class as any other, so it has the freedom to inherit from whatever other classes it likes. In time, you won't think twice about such inheritance relationships.

Automating Reference Count Manipulations

The `RCObject` class gives us a place to store a reference count, and it gives us member functions through which that reference count can be manipulated, but the *calls* to those functions must still be

manually inserted in other classes. It is still up to the `String` copy constructor and the `String` assignment operator to call `addReference` and `removeReference` on `StringValue` objects. This is clumsy. We'd like to move those calls out into a reusable class, too, thus freeing authors of classes like `String` from worrying about *any* of the details of reference counting. Can it be done? Isn't C++ supposed to support reuse?

It can, and it does. There's no easy way to arrange things so that *all* reference-counting considerations can be moved out of application classes, but there is a way to eliminate *most* of them for most classes. (In some application classes, you *can* eliminate all reference-counting code, but our `String` class, alas, isn't one of them. One member function spoils the party, and I suspect you won't be too surprised to hear it's our old nemesis, the non-`const` version of `operator[]`. Take heart, however; we'll tame that miscreant in the end.)

Notice that each `String` object contains a pointer to the `StringValue` object representing that `String`'s value:

```
class String {
private:
  struct StringValue: public RCObject { ... };


  StringValue *value;                // value of this String


  ...


};
```

We have to manipulate the `refCount` field of the `StringValue` object anytime anything interesting happens to one of the pointers pointing to it. "Interesting happenings" include copying a pointer, reassigning one, and destroying one. If we could somehow make the *pointer itself* detect these happenings and automatically perform the necessary manipulations of the `refCount` field, we'd be home free. Unfortunately, pointers are rather dense creatures, and the chances of them detecting anything, much less automatically reacting to things they detect, are pretty slim. Fortunately, there's a way to smarten them up: replace them with objects that *act like* pointers, but that do more.

Such objects are called *smart pointers*, and you can read about them in more detail than you probably care to in Item 28. For our purposes here, it's enough to know that smart pointer objects support the member selection (`->`) and dereferencing (`*`) operations, just like real pointers (which, in this context, are generally referred to as *dumb pointers*), and, like dumb pointers, they are strongly typed: you can't make a smart pointer-to-T point to an object that isn't of type T.

Here's a template for objects that act as smart pointers to reference-counted objects:

```
// template class for smart pointers-to-T objects. T must
// support the RCObject interface, typically by inheriting
// from RCObject
template<class T>
class RCPtr {
public:
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();


  RCPtr& operator=(const RCPtr& rhs);



  T* operator->() const;            // see Item 28
  T& operator*() const;             // see Item 28


private:
  T *pointee;                       // dumb pointer this
                                    // object is emulating



  void init();                      // common initialization
};                                  // code
```

This template gives smart pointer objects control over what happens during their construction, assignment, and destruction. When such events occur, these objects can automatically perform the appropriate manipulations of the `refCount` field in the objects to which they point.

For example, when an `RCPtr` is created, the object it points to needs to have its reference count increased. There's no need to burden application developers with the requirement to tend to this irksome detail manually, because `RCPtr` constructors can handle it themselves. The code in the two constructors is all but identical — only the member initialization lists differ — so rather than write it twice, we put it in a private member function called `init` and have both constructors call that:

```
template<class T>
RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
{
  init();
}


template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
{
  init();
```

```
}


template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) {               // if the dumb pointer is
    return;                         // null, so is the smart one
  }



  if (pointee->isShareable() == false) {         // if the value
    pointee = new T(*pointee);                   // isn't shareable,
  }                                              // copy it



  pointee->addReference();          // note that there is now a
}                                   // new reference to the value
```

Moving common code into a separate function like `init` is exemplary software engineering, but its luster dims when, as in this case, the function doesn't behave correctly.

The problem is this. When `init` needs to create a new copy of a value (because the existing copy isn't shareable), it executes the following code:

```
pointee = new T(*pointee);
```

The type of `pointee` is pointer-to-`T`, so this statement creates a new `T` object and initializes it by calling `T`'s copy constructor. In the case of an `RCPtr` in the `String` class, `T` will be `String::StringValue`, so the statement above will call `String::StringValue`'s copy constructor. We haven't declared a copy constructor for that class, however, so our compilers will generate one for us. The copy constructor so generated will, in accordance with the rules for automatically generated copy constructors in C++, copy only `StringValue`'s `data` *pointer*; it will *not* copy the `char*` string `data` points to. Such behavior is disastrous in nearly *any* class (not just reference-counted classes), and that's why you should get into the habit of writing a copy constructor (and an assignment operator) for all your classes that contain pointers (see Item E11).

The correct behavior of the `RCPtr<T>` template depends on `T` containing a copy constructor that makes a truly independent copy (i.e., a *deep copy*) of the value represented by `T`. We must augment `StringValue` with such a constructor before we can use it with the `RCPtr` class:

```
class String {
private:
```

```
  struct StringValue: public RCObject {
    StringValue(const StringValue& rhs);


     ...


  };


  ...


};


String::StringValue::StringValue(const StringValue& rhs)
{
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
}
```

The existence of a deep-copying copy constructor is not the only assumption `RCPtr<T>` makes about `T`. It also requires that `T` inherit from `RCObject`, or at least that `T` provide all the functionality that `RCObject` does. In view of the fact that `RCPtr` objects are designed to point only to reference-counted objects, this is hardly an unreasonable assumption. Nevertheless, the assumption must be documented.

A final assumption in `RCPtr<T>` is that the type of the object pointed to is `T`. This seems obvious enough. After all, `pointee` is declared to be of type `T*`. But `pointee` might really point to a class *derived* from `T`. For example, if we had a class `SpecialStringValue` that inherited from `String::StringValue`,

```
class String {
private:
  struct StringValue: public RCObject { ... };


  struct SpecialStringValue: public StringValue { ... };


  ...


};
```

we could end up with a `String` containing a `RCPtr<StringValue>` pointing to a `SpecialStringValue` object. In that case, we'd want this part of `init`,

```
pointee = new T(*pointee);                // T is StringValue, but
                                          // pointee really points to
                                          // a SpecialStringValue
```

to call `SpecialStringValue`'s copy constructor, not `StringValue`'s. We can arrange for this to happen by using a virtual copy constructor (see Item 25). In the case of our `String` class, we don't expect classes to derive from `StringValue`, so we'll disregard this issue.

With `RCPtr`'s constructors out of the way, the rest of the class's functions can be dispatched with considerably greater alacrity. Assignment of an `RCPtr` is straightforward, though the need to test whether the newly assigned value is shareable complicates matters slightly. Fortunately, such complications have already been handled by the `init` function that was created for `RCPtr`'s constructors. We take advantage of that fact by using it again here:

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {          // skip assignments
                                         // where the value
                                         // doesn't change
    if (pointee) {
      pointee->removeReference();        // remove reference to
    }                                    // current value



    pointee = rhs.pointee;               // point to new value
    init();                              // if possible, share it
  }                                      // else make own copy


  return *this;
}
```

The destructor is easier. When an `RCPtr` is destroyed, it simply removes its reference to the reference-counted object:

```
template<class T>
RCPtr<T>::~RCPtr()
{
  if (pointee)pointee->removeReference();
}
```

If the `RCPtr` that just expired was the last reference to the object, that object will be destroyed inside `RCObject`'s `removeReference` member function. Hence `RCPtr` objects never need to worry about destroying the values they point to.

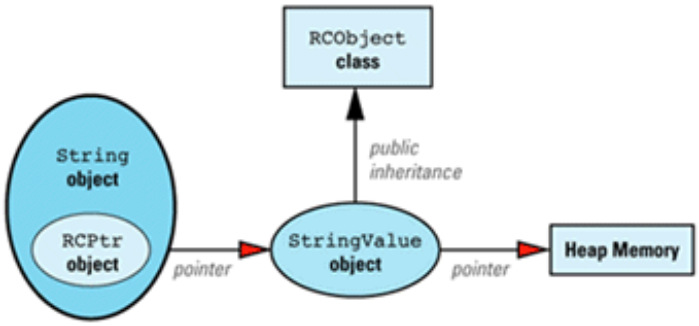
Finally, `RCPtr`'s pointer-emulating operators are part of the smart pointer boilerplate you can read about in Item 28:

```
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }


template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

Putting it All Together

Enough! *Finis*! At long last we are in a position to put all the pieces together and build a reference-counted `String` class based on the reusable `RCObject` and `RCPtr` classes. With luck, you haven't forgotten that that was our original goal.

Each reference-counted string is implemented via this data structure:



The classes making up this data structure are defined like this:

```
template<class T>                       // template class for smart
class RCPtr {                           // pointers-to-T objects; T
public:                                 // must inherit from RCObject
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
```

```
  ~RCPtr();

  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;
  T& operator*() const;

private:
  T *pointee;

  void init();
};


class RCObject {                        // base class for reference-
public:                                 // counted objects
  void addReference();
  void removeReference();

  void markUnshareable();
  bool isShareable() const;

  bool isShared() const;

protected:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;

private:
  int refCount;
  bool shareable;
};

class String {                            // class to be used by
public:                                   // application developers

  String(const char *value = "");

  const char& operator[](int index) const;
  char& operator[](int index);

private:
  // class representing string values
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    StringValue(const StringValue& rhs);
    void init(const char *initValue);
    ~StringValue();
  };

  RCPtr<StringValue> value;
};
```

For the most part, this is just a recap of what we've already developed, so nothing should be much of a surprise. Close examination reveals we've added an `init` function to `String::StringValue`, but, as we'll see below, that serves the same purpose as the corresponding function in `RCPtr`: it prevents code duplication in the constructors.

There is a significant difference between the public interface of this `String` class and the one we used at the beginning of this Item. Where is the copy constructor? Where is the assignment operator? Where is the destructor? Something is definitely amiss here.

Actually, no. Nothing is amiss. In fact, some things are working perfectly. If you don't see what they are, prepare yourself for a C++ epiphany.

We don't need those functions anymore. Sure, copying of `String` objects is still supported, and yes, the copying will correctly handle the underlying reference-counted `StringValue` objects, but the `String` class doesn't have to provide a single line of code to make this happen. That's because the compiler-generated copy constructor for `String` will automatically call the copy constructor for `String`'s `RCPtr` member, and the copy constructor for that class will perform all the necessary manipulations of the `StringValue` object, including its reference count. An `RCPtr` is a *smart* pointer, remember? We designed it to take care of the details of reference counting, so that's what it does. It also handles assignment and destruction, and that's why `String` doesn't need to write those functions, either. Our original goal was to move the unreusable reference-counting code out of our hand-written `String` class and into context-independent classes where it would be available for use with *any* class. Now we've done it (in the form of the `RCObject` and `RCPtr` classes), so don't be so surprised when it suddenly starts working. It's *supposed* to work.

Just so you have everything in one place, here's the implementation of `RCObject`:

```
RCObject::RCObject()
: refCount(0), shareable(true) {}


RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}


RCObject& RCObject::operator=(const RCObject&)
{ return *this; }


RCObject::~RCObject() {}


void RCObject::addReference() { ++refCount; }
```

```
void RCObject::removeReference()
{ if (--refCount == 0) delete this; }


void RCObject::markUnshareable()
{ shareable = false; }


bool RCObject::isShareable() const
{ return shareable; }


bool RCObject::isShared() const
{ return refCount > 1; }
```

And here's the implementation of `RCPtr`:

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) return;



  if (pointee->isShareable() == false) {
    pointee = new T(*pointee);
  }


  pointee->addReference();
}


template<class T>
RCPtr<T>::RCPtr(T* realPtr)
: pointee(realPtr)
{ init(); }


template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs)
: pointee(rhs.pointee)
{ init(); }


template<class T>
RCPtr<T>::~RCPtr()
{ if (pointee)pointee->removeReference(); }
```

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {
    if (pointee) pointee->removeReference();



    pointee = rhs.pointee;
    init();
  }



  return *this;
}



template<class T>
T* RCPtr<T>::operator->() const { return pointee; }



template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

The implementation of `String::StringValue` looks like this:

```
void String::StringValue::init(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}



String::StringValue::StringValue(const char *initValue)
{ init(initValue); }



String::StringValue::StringValue(const StringValue& rhs)
{ init(rhs.data); }



String::StringValue::~StringValue()
{ delete [] data; }
```

Ultimately, all roads lead to `String`, and that class is implemented this way:

```
String::String(const char *initValue)
: value(new StringValue(initValue)) {}


const char& String::operator[](int index) const
{ return value->data[index]; }


char& String::operator[](int index)
{
  if (value->isShared()) {
    value = new StringValue(value->data);
  }


  value->markUnshareable();


  return value->data[index];
}
```

If you compare the code for this `String` class with that we developed for the `String` class using dumb pointers, you'll be struck by two things. First, there's a lot less of it here than there. That's because `RCPtr` has assumed much of the reference-counting burden that used to fall on `String`. Second, the code that remains in `String` is nearly unchanged: the smart pointer replaced the dumb pointer essentially seamlessly. In fact, the only changes are in `operator[]`, where we call `isShared` instead of checking the value of `refCount` directly and where our use of the smart `RCPtr` object eliminates the need to manually manipulate the reference count during a copy-on-write.

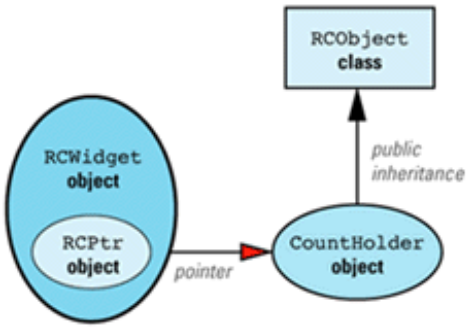
This is all very nice, of course. Who can object to less code? Who can oppose encapsulation success stories? The bottom line, however, is determined more by the impact of this newfangled `String` class on its clients than by any of its implementation details, and it is here that things really shine. If no news is good news, the news here is very good indeed. *The `String` interface has not changed*. We added reference counting, we added the ability to mark individual string values as unshareable, we moved the notion of reference countability into a new base class, we added smart pointers to automate the manipulation of reference counts, yet not one line of client code needs to be changed. Sure, we changed the `String` class definition, so clients who want to take advantage of reference-counted strings must recompile and relink, but their investment in code is completely and utterly preserved. You see? Encapsulation really *is* a wonderful thing.

Adding Reference Counting to Existing Classes

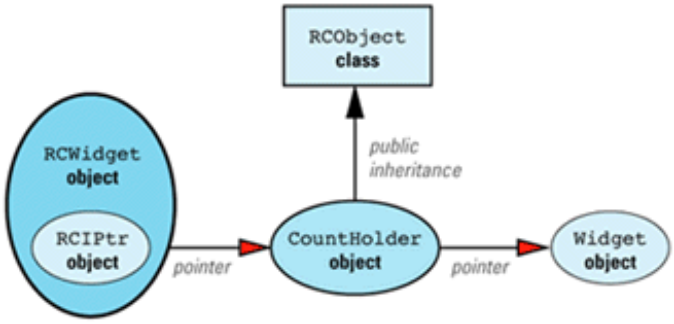
Everything we've discussed so far assumes we have access to the source code of the classes we're interested in. But what if we'd like to apply the benefits of reference counting to some class `Widget` that's in a library we can't modify? There's no way to make `Widget` inherit from `RCObject`, so we can't use smart `RCPtrs` with it. Are we out of luck?

We're not. With some minor modifications to our design, we can add reference counting to *any* type.

First, let's consider what our design would look like if we could have `Widget` inherit from `RCObject`. In that case, we'd have to add a class, `RCWidget`, for clients to use, but everything would then be analogous to our `String`/`StringValue` example, with `RCWidget` playing the role of `String` and `Widget` playing the role of `StringValue`. The design would look like this:



We can now apply the maxim that most problems in Computer Science can be solved with an additional level of indirection. We add a new class, `CountHolder`, to hold the reference count, and we have `CountHolder` inherit from `RCObject`. We also have `CountHolder` contain a pointer to a `Widget`. We then replace the smart `RCPtr` template with an equally smart `RCIPtr` template that knows about the existence of the `CountHolder` class. (The "I" in `RCIPtr` stands for "indirect.") The modified design looks like this:



Just as `StringValue` was an implementation detail hidden from clients of `String`, `CountHolder` is an implementation detail hidden from clients of `RCWidget`. In fact, it's an implementation detail of `RCIPtr`, so it's nested inside that class. `RCIPtr` is implemented this way:

```
template<class T>
class RCIPtr {
public:
  RCIPtr(T* realPtr = 0);
  RCIPtr(const RCIPtr& rhs);
  ~RCIPtr();

  RCIPtr& operator=(const RCIPtr& rhs);

  const T* operator->() const;              // see below for an
  T* operator->();                          // explanation of why
  const T& operator*() const;               // these functions are
  T& operator*();                           // declared this way

private:
  struct CountHolder: public RCObject {
    ~CountHolder() { delete pointee; }
    T *pointee;
  };

  CountHolder *counter;

  void init();
  void makeCopy();                          // see below
};

template<class T>
void RCIPtr<T>::init()
{
  if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
  }

  counter->addReference();
}

template<class T>
RCIPtr<T>::RCIPtr(T* realPtr)
: counter(new CountHolder)
{
  counter->pointee = realPtr;
  init();
}

template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs)
: counter(rhs.counter)
{ init(); }
```

```
template<class T>
RCIPtr<T>::~RCIPtr()
{ counter->removeReference(); }

template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
{
  if (counter != rhs.counter) {
    counter->removeReference();
    counter = rhs.counter;
    init();
  }
  return *this;
}

template<class T>                          // implement the copy
void RCIPtr<T>::makeCopy()                 // part of copy-on-
{                                          // write (COW)
  if (counter->isShared()) {
    T *oldValue = counter->pointee;
    counter->removeReference();
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
    counter->addReference();
  }
}

template<class T>                           // const access;
const T* RCIPtr<T>::operator->() const      // no COW needed
{ return counter->pointee; }

template<class T>                           // non-const
T* RCIPtr<T>::operator->()                  // access; COW
{ makeCopy(); return counter->pointee; }    // needed

template<class T>                           // const access;
const T& RCIPtr<T>::operator*() const       // no COW needed
{ return *(counter->pointee); }

template<class T>                           // non-const
T& RCIPtr<T>::operator*()                   // access; do the
{ makeCopy(); return *(counter->pointee); } // COW thing
```

`RCIPtr` differs from `RCPtr` in only two ways. First, `RCPtr` objects point to values directly, while `RCIPtr` objects point to values through intervening `CountHolder` objects. Second, `RCIPtr` overloads `operator->` and `operator*` so that a copy-on-write is automatically performed whenever a non-`const` access is made to a pointed-to object.

Given `RCIPtr`, it's easy to implement `RCWidget`, because each function in `RCWidget` is implemented by forwarding the call through the underlying `RCIPtr` to a `Widget` object. For example, if `Widget` looks like this,

```
class Widget {
```

```
public:
  Widget(int size);
  Widget(const Widget& rhs);
  ~Widget();
  Widget& operator=(const Widget& rhs);


  void doThis();
  int showThat() const;
};
```

`RCWidget` will be defined this way:

```
class RCWidget {
public:
  RCWidget(int size): value(new Widget(size)) {}
  void doThis() { value->doThis(); }
  int showThat() const { return value->showThat(); }


private:
  RCIPtr<Widget> value;
};
```

Note how the `RCWidget` constructor calls the `Widget` constructor (via the `new` operator — see Item 8) with the argument it was passed; how `RCWidget`'s `doThis` calls `doThis` in the `Widget` class; and how `RCWidget::showThat` returns whatever its `Widget` counterpart returns. Notice also how `RCWidget` declares no copy constructor, no assignment operator, and no destructor. As with the `String` class, there is no need to write these functions. Thanks to the behavior of the `RCIPtr` class, the default versions do the right things.

If the thought occurs to you that creation of `RCWidget` is so mechanical, it could be automated, you're right. It would not be difficult to write a program that takes a class like `Widget` as input and produces a class like `RCWidget` as output. If you write such a program, please let me know.

Evaluation

Let us disentangle ourselves from the details of widgets, strings, values, smart pointers, and reference-counting base classes. That gives us an opportunity to step back and view reference counting in a broader context. In that more general context, we must address a higher-level question, namely, when is reference counting an appropriate technique?

Reference-counting implementations are not without cost. Each reference-counted value carries a reference count with it, and most operations require that this reference count be examined or manipulated in some way. Object values therefore require more memory, and we sometimes execute more code when we work with them. Furthermore, the underlying source code is considerably more complex for a reference-counted class than for a less elaborate implementation.

An un-reference-counted string class typically stands on its own, while our final `String` class is useless unless it's augmented with three auxiliary classes (`StringValue`, `RCObject`, and `RCPtr`). True, our more complicated design holds out the promise of greater efficiency when values can be shared, it eliminates the need to track object ownership, and it promotes reusability of the reference counting idea and implementation. Nevertheless, that quartet of classes has to be written, tested, documented, and maintained, and that's going to be more work than writing, testing, documenting, and maintaining a single class. Even a manager can see that.

Reference counting is an optimization technique predicated on the assumption that objects will commonly share values (see also Item 18). If this assumption fails to hold, reference counting will use more memory than a more conventional implementation and it will execute more code. On the other hand, if your objects *do* tend to have common values, reference counting should save you both time and space. The bigger your object values and the more objects that can simultaneously share values, the more memory you'll save. The more you copy and assign values between objects, the more time you'll save. The more expensive it is to create and destroy a value, the more time you'll save there, too. In short, reference counting is most useful for improving efficiency under the following conditions:

- **Relatively few values are shared by relatively many objects.** Such sharing typically arises through calls to assignment operators and copy constructors. The higher the objects/values ratio, the better the case for reference counting.
- **Object values are expensive to create or destroy, or they use lots of memory.** Even when this is the case, reference counting still buys you nothing unless these values can be shared by multiple objects.

There is only one sure way to tell whether these conditions are satisfied, and that way is *not* to guess or rely on your programmer's intuition (see Item 16). The reliable way to find out whether your program can benefit from reference counting is to profile or instrument it. That way you can find out if creating and destroying values is a performance bottleneck, and you can measure the objects/values ratio. Only when you have such data in hand are you in a position to determine whether the benefits of reference counting (of which there are many) outweigh the disadvantages (of which there are also many).

Even when the conditions above are satisfied, a design employing reference counting may still be inappropriate. Some data structures (e.g., directed graphs) lead to self-referential or circular dependency structures. Such data structures have a tendency to spawn isolated collections of objects, used by no one, whose reference counts never drop to zero. That's because each object in the unused structure is pointed to by at least one other object in the same structure. Industrial-strength garbage collectors use special techniques to find such structures and eliminate them, but the simple reference-counting approach we've examined here is not easily extended to include such techniques.

Reference counting can be attractive even if efficiency is not your primary concern. If you find yourself weighed down with uncertainty over who's allowed to delete what, reference counting could be just the technique you need to ease your burden. Many programmers are devoted to reference counting for this reason alone.

Let us close this discussion on a technical note by tying up one remaining loose end. When `RCObject::removeReference` decrements an object's reference count, it checks to see if the new count is 0. If it is, `removeReference` destroys the object by `delete`ing `this`. This is a safe operation only if the object was allocated by calling `new`, so we need some way of ensuring that `RCObjects` are created only in that manner.

In this case we do it by convention. `RCObject` is designed for use as a base class of reference-counted value objects, and those value objects should be referred to only by smart `RCPtr` pointers. Furthermore, the value objects should be instantiated only by application objects that realize values are being shared; the classes describing the value objects should never be available for general use. In our example, the class for value objects is `StringValue`, and we limit its use by making it private in `String`. Only `String` can create `StringValue` objects, so it is up to the author of the `String` class to ensure that all such objects are allocated via `new`.

Our approach to the constraint that `RCObjects` be created only on the heap, then, is to assign responsibility for conformance to this constraint to a well-defined set of classes and to ensure that only that set of classes can create `RCObjects`. There is no possibility that random clients can accidently (or maliciously) create `RCObjects` in an inappropriate manner. We limit the right to create reference-counted objects, and when we do hand out the right, we make it clear that it's accompanied by the concomitant responsibility to follow the rules governing object creation.

Back to Item 28: Smart pointers
Continue to Item 30: Proxy classes

---

[10] The `string` type in the standard C++ library (see Item E49 and Item 35) uses a combination of solutions two and three. The reference returned from the non-`const` `operator[]` is guaranteed to be valid until the next function call that might modify the string. After that, use of the reference (or the character to which it refers) yields undefined results. This allows the string's shareability flag to be reset to `true` whenever a function is called that might modify the string.
Return

Item 30: Proxy classes.

Though your in-laws may be one-dimensional, the world, in general, is not. Unfortunately, C++ hasn't yet caught on to that fact. At least, there's little evidence for it in the language's support for arrays. You can create two-dimensional, three-dimensional — heck, you can create n-dimensional — arrays in FORTRAN, in BASIC, even in COBOL (okay, FORTRAN only allows up to seven dimensions, but let's not quibble), but can you do it in C++? Only sometimes, and even then only sort of.

This much is legal:

```
int data[10][20];                          // 2D array: 10 by 20
```

The corresponding construct using variables as dimension sizes, however, is not:

```
void processInput(int dim1, int dim2)
{
  int data[dim1][dim2];                    // error! array dimensions
  ...                                      // must be known during
}                                          // compilation
```

It's not even legal for a heap-based allocation:

```
int *data =
  new int[dim1][dim2];                     // error!
```

Implementing Two-Dimensional Arrays

Multidimensional arrays are as useful in C++ as they are in any other language, so it's important to come up with a way to get decent support for them. The usual way is the standard one in C++: create a class to represent the objects we need but that are missing in the language proper. Hence we can define a class template for two-dimensional arrays:

```
template<class T>
class Array2D {
public:
  Array2D(int dim1, int dim2);
  ...

};
```

Now we can define the arrays we want:

```
Array2D<int> data(10, 20);             // fine
```

```
Array2D<float> *data =
  new Array2D<float>(10, 20);          // fine

void processInput(int dim1, int dim2)
{
  Array2D<int> data(dim1, dim2);       // fine
  ...
}
```

Using these array objects, however, isn't quite as straightforward. In keeping with the grand syntactic tradition of both C and C++, we'd like to be able to use brackets to index into our arrays,

```
cout << data[3][6];
```

but how do we declare the indexing operator in `Array2D` to let us do this?

Our first impulse might be to declare `operator[][]` functions, like this:

```
template<class T>
class Array2D {
public:


  // declarations that won't compile
  T& operator[][](int index1, int index2);
  const T& operator[][](int index1, int index2) const;


  ...


};
```

We'd quickly learn to rein in such impulses, however, because there is no such thing as `operator[][]` , and don't think your compilers will forget it. (For a complete list of operators, overloadable and otherwise, see Item 7 .) We'll have to do something else.

If you can stomach the syntax, you might follow the lead of the many programming languages that use parentheses to index into arrays. To use parentheses, you just overload `operator()` :

```
template<class T>
class Array2D {
public:


  // declarations that will compile
  T& operator()(int index1, int index2);
  const T& operator()(int index1, int index2) const;


  ...
```

```
};
```

Clients then use arrays this way:

```
cout << data(3, 6);
```

This is easy to implement and easy to generalize to as many dimensions as you like. The drawback is that your `Array2D` objects don't look like built-in arrays any more. In fact, the above access to element (3, 6) of `data` looks, on the face of it, like a function call.

If you reject the thought of your arrays looking like FORTRAN refugees, you might turn again to the notion of using brackets as the indexing operator. Although there is no such thing as `operator[][]` , it is nonetheless legal to write code that appears to use it:

```
int data[10][20];


...


cout << data[3][6];          // fine
```

What gives?

What gives is that the variable `data` is not really a two-dimensional array at all, it's a 10-element one-dimensional array. Each of those 10 elements is itself a 20-element array, so the expression `data[3][6]` really means `(data[3])[6]` , i.e., the seventh element of the array that is the fourth element of `data` . In short, the value yielded by the first application of the brackets is another array, so the second application of the brackets gets an element from that secondary array.

We can play the same game with our `Array2D` class by overloading `operator[]` to return an object of a new class, `Array1D` . We can then overload `operator[]` again in `Array1D` to return an element in our original two-dimensional array:

```
template<class T>
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](int index);
    const T& operator[](int index) const;
    ...

  };

  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...

};
```

The following then becomes legal:

```
Array2D<float> data(10, 20);

...

cout << data[3][6];          // fine
```

Here, `data[3]` yields an `Array1D` object and the `operator[]` invocation on that object yields the float in position (3, 6) of the original two-dimensional array.

Clients of the `Array2D` class need not be aware of the presence of the `Array1D` class. Objects of this latter class stand for one-dimensional array objects that, conceptually, do not exist for clients of `Array2D` . Such clients program as if they were using real, live, honest-to-Allah two-dimensional arrays. It is of no concern to `Array2D` clients that those objects must, in order to satisfy the vagaries of C++, be syntactically compatible with one-dimensional arrays of other one-dimensional arrays.

Each `Array1D` object *stands for* a one-dimensional array that is absent from the conceptual model used by clients of `Array2D` . Objects that stand for other objects are often called *proxy objects* , and the classes that give rise to proxy objects are often called *proxy classes* . In this example, `Array1D` is a proxy class. Its instances stand for one-dimensional arrays that, conceptually, do not exist. (The terminology for proxy objects and classes is far from universal; objects of such classes are also sometimes known as *surrogates* .)

Distinguishing Reads from Writes via `operator[]`

The use of proxies to implement classes whose instances act like multidimensional arrays is common, but proxy classes are more flexible than that. Item 5 , for example, shows how proxy classes can be employed to prevent single-argument constructors from being used to perform unwanted type conversions. Of the varied uses of proxy classes, however, the most heralded is that of helping distinguish reads from writes through `operator[]` .

Consider a reference-counted string type that supports `operator[]` . Such a type is examined in detail in Item 29 . If the concepts behind reference counting have slipped your mind, it would be a good idea to familiarize yourself with the material in that Item now.

A string type supporting `operator[]` allows clients to write code like this:

```
String s1, s2;           // a string-like class; the
                         // use of proxies keeps this
                         // class from conforming to
                         // the standard string
...                      // interface

cout << s1[5];           // read s1
```

```
s2[5] = 'x';              // write s2

s1[3] = s2[8];            // write s1, read s2
```

Note that `operator[]` can be called in two different contexts: to read a character or to write a character. Reads are known as *rvalue* usages; writes are known as *lvalue* usages. (The terms come from the field of compilers, where an lvalue goes on the left-hand side of an assignment and an rvalue goes on the right-hand side.) In general, using an object as an lvalue means using it such that it might be modified, and using it as an rvalue means using it such that it cannot be modified.

We'd like to distinguish between lvalue and rvalue usage of `operator[]` because, especially for reference-counted data structures, reads can be much less expensive to implement than writes. As Item 29 ex plains, writes of reference-counted objects may involve copying an entire data structure, but reads never require more than the simple returning of a value. Unfortunately, inside `operator[]` , there is no way to determine the context in which the function was called; it is not possible to distinguish lvalue usage from rvalue usage within `operator[]` .

"But wait," you say, "we don't need to. We can overload `operator[]` on the basis of its `const` ness, and that will allow us to distinguish reads from writes." In other words, you suggest we solve our problem this way:

```
class String {
public:
  const char& operator[](int index) const;       // for reads
  char& operator[](int index);                    // for writes
  ...

};
```

Alas, this won't work. Compilers choose between `const` and non-`const` member functions by looking only at whether the *object* invoking a function is `const` . No consideration is given to the context in which a call is made. Hence:

```
String s1, s2;

...

cout << s1[5];                        // calls non-const operator[],
                                      // because s1 isn't const

s2[5] = 'x';                          // also calls non-const
                                      // operator[]: s2 isn't const

s1[3] = s2[8];                        // both calls are to non-const
                                      // operator[], because both s1
                                      // and s2 are non-const objects
```

Overloading `operator[]` , then, fails to distinguish reads from writes.

In Item 29 , we resigned ourselves to this unsatisfactory state of affairs and made the conservative assumption that all calls to `operator[]` were for writes. This time we shall not give up so easily. It may be impossible to distinguish lvalue from rvalue usage inside `operator[]` , but we still want to do it. We will therefore find a way. What fun is life if you allow yourself to be limited by the possible?

Our approach is based on the fact that though it may be impossible to tell whether `operator[]` is being invoked in an lvalue or an rvalue context from within `operator[]` , we can still treat reads differently from writes if we *delay* our lvalue-versus-rvalue actions until we see how the result of `operator[]` is used. All we need is a way to postpone our decision on whether our object is being read or written until *after* `operator[]` has returned. (This is an example of *lazy evaluation* — see Item 17 .)

A proxy class allows us to buy the time we need, because we can modify `operator[]` to return a *proxy* for a string character instead of a string character itself. We can then wait to see how the proxy is used. If it's read, we can belatedly treat the call to `operator[]` as a read. If it's written, we must treat the call to `operator[]` as a write.

We will see the code for this in a moment, but first it is important to understand the proxies we'll be using. There are only three things you can do with a proxy:

- Create it, i.e., specify which string character it stands for.
- Use it as the target of an assignment, in which case you are really making an assignment to the string character it stands for. When used in this way, a proxy represents an lvalue use of the string on which `operator[]` was invoked.
- Use it in any other way. When used like this, a proxy represents an rvalue use of the string on which `operator[]` was invoked.

Here are the class definitions for a reference-counted `String` class using a proxy class to distinguish between lvalue and rvalue usages of `operator[]` :

```
class String {                    // reference-counted strings;
public:                           // see Item 29 for details

  class CharProxy {               // proxies for string chars
  public:
    CharProxy(String& str, int index);               // creation

    CharProxy& operator=(const CharProxy& rhs);      // lvalue
    CharProxy& operator=(char c);                    // uses

    operator char() const;                           // rvalue
                                                     // use
  private:
    String& theString;            // string this proxy pertains to

    int charIndex;                // char within that string
                                  // this proxy stands for
  };
```

```
  // continuation of String class
  const CharProxy
    operator[](int index) const;   // for const Strings


  CharProxy operator[](int index); // for non-const Strings
  ...

friend class CharProxy;

private:
  RCPtr<StringValue> value;
};
```

Other than the addition of the `CharProxy` class (which we'll examine below), the only difference between this `String` class and the final `String` class in Item 29 is that both `operator[]` functions now return `CharProxy` objects. Clients of `String` can generally ignore this, however, and program as if the `operator[]` functions returned characters (or references to characters — see Item 1) in the usual manner:

```
String s1, s2;          // reference-counted strings
                        // using proxies
...

cout << s1[5];          // still legal, still works

s2[5] = 'x';            // also legal, also works

s1[3] = s2[8];          // of course it's legal,
                        // of course it works
```

What's interesting is not that this works. What's interesting is *how* it works.

Consider first this statement:

```
cout << s1[5];
```

The expression `s1[5]` yields a `CharProxy` object. No output operator is defined for such objects, so your compilers labor to find an implicit type conversion they can apply to make the call to `operator<<` succeed (see Item 5). They find one: the implicit conversion from `CharProxy` to `char` declared in the `CharProxy` class. They automatically invoke this conversion operator, and the result is that the string character represented by the `CharProxy` is printed. This is representative of the `CharProxy` -to-`char` conversion that takes place for all `CharProxy` objects used as rvalues.

Lvalue usage is handled differently. Look again at

```
s2[5] = 'x';
```

As before, the expression `s2[5]` yields a `CharProxy` object, but this time that object is the target of an assignment. Which assignment operator is invoked? The target of the assignment is a `CharProxy` , so the assignment operator that's called is in the `CharProxy` class. This is crucial, because inside a `CharProxy` assignment operator, we know that the `CharProxy` object being assigned to is being used as an lvalue. We therefore know that the string character for which the proxy stands is being used as an lvalue, and we must take whatever actions are necessary to implement lvalue access for that character.

Similarly, the statement

```
s1[3] = s2[8];
```

calls the assignment operator for two `CharProxy` objects, and inside that operator we know the object on the left is being used as an lvalue and the object on the right as an rvalue.

"Yeah, yeah, yeah," you grumble, "show me." Okay. Here's the code for `String` 's `operator[]` functions:

```
const String::CharProxy String::operator[](int index) const
{
  return CharProxy(const_cast<String&>(*this), index);
}


String::CharProxy String::operator[](int index)
{
  return CharProxy(*this, index);
}
```

Each function just creates and returns a proxy for the requested character. No action is taken on the character itself: we defer such action until we know whether the access is for a read or a write.

Note that the `const` version of `operator[]` returns a `const` proxy. Because `CharProxy::operator=` isn't a `const` member function, such proxies can't be used as the target of assignments. Hence neither the proxy returned from the `const` version of `operator[]` nor the character for which it stands may be used as an lvalue. Conveniently enough, that's exactly the behavior we want for the `const` version of `operator[]` .

Note also the use of a `const_cast` (see Item 2) on `*this` when creating the `CharProxy` object that the `const` `operator[]` returns. That's necessary to satisfy the constraints of the `CharProxy` constructor, which accepts only a non-`const` `String` . Casts are usually worrisome, but in this case the `CharProxy` object returned by `operator[]` is itself `const` , so there is no risk the `String` containing the character to which the proxy refers will be modified.

Each proxy returned by an `operator[]` function remembers which string it pertains to and, within that string, the index of the character it represents:

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

Conversion of a proxy to an rvalue is straightforward — we just return a copy of the character represented by the proxy:

```
String::CharProxy::operator char() const
{
  return theString.value->data[charIndex];
}
```

If you've forgotten the relationship among a `String` object, its `value` member, and the `data` member it points to, you can refresh your memory by turning to Item 29 . Because this function returns a character by value, and because C++ limits the use of such by-value returns to rvalue contexts only, this conversion function can be used only in places where an rvalue is legal.

We thus turn to implementation of `CharProxy` 's assignment operators, which is where we must deal with the fact that a character represented by a proxy is being used as the target of an assignment, i.e., as an lvalue. We can implement `CharProxy` 's conventional assignment operator as follows:

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
  // if the string is sharing a value with other String objects,
  // break off a separate copy of the value for this string only
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // now make the assignment: assign the value of the char
  // represented by rhs to the char represented by *this
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];

  return *this;
}
```

If you compare this with the implementation of the non-`const String` ::`operator` in Item 29 , you'll see that they are strikingly similar. This is to be expected. In Item 29 , we pessimistically assumed that all invocations of the non-`const operator[]` were writes, so we treated them as such. Here, we moved the code implementing a write into `CharProxy` 's assignment operators, and that allows us to avoid paying for a write when the non-`const operator[]` is used only in an rvalue context. Note, by the way, that this function requires access to `String` 's private data member `value.` That's why `CharProxy` is declared a friend in the earlier class definition for `String` .

The second `CharProxy` assignment operator is almost identical:

```
String::CharProxy& String::CharProxy::operator=(char c)
{
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
```

```
    }

    theString.value->data[charIndex] = c;

    return *this;
  }
```

As an accomplished software engineer, you would, of course, banish the code duplication present in these two assignment operators to a private `CharProxy` member function that both would call. Aren't you the modular one?

Limitations

The use of a proxy class is a nice way to distinguish lvalue and rvalue usage of `operator[]` , but the technique is not without its drawbacks. We'd like proxy objects to seamlessly replace the objects they stand for, but this ideal is difficult to achieve. That's because objects are used as lvalues in contexts other than just assignment, and using proxies in such contexts usually yields behavior different from using real objects.

Consider again the code fragment from Item 29 that motivated our decision to add a shareability flag to each `StringValue` object. If `String` ::`operator[]` returns a `CharProxy` instead of a `char&` , that code will no longer compile:

```
String s1 = "Hello";


char *p = &s1[1];            // error!
```

The expression `s1[1]` returns a `CharProxy` , so the type of the expression on the right-hand side of the "= " is `CharProxy*` . There is no conversion from a `CharProxy*` to a `char*` , so the initialization of `p` fails to compile. In general, taking the address of a proxy yields a different type of pointer than does taking the address of a real object.

To eliminate this difficulty, you'll need to overload the address-of operators for the `CharProxy` class:

```
class String {
public:


  class CharProxy {
  public:
    ...
    char * operator&();
    const char * operator&() const;
    ...
  };


  ...
};
```

These functions are easy to implement. The `const` function just returns a pointer to a `const` version of the character represented by the proxy:

```
const char * String::CharProxy::operator&() const
{
  return &(theString.value->data[charIndex]);
}
```

The non-`const` function is a bit more work, because it returns a pointer to a character that may be modified. This is analogous to the behavior of the non-`const` version of `String` ::`operator[]` in Item 29 , and the implementation is equally analogous:

```
char * String::CharProxy::operator&()
{
  // make sure the character to which this function returns
  // a pointer isn't shared by any other String objects
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // we don't know how long the pointer this function
  // returns will be kept by clients, so the StringValue
  // object can never be shared
  theString.value->markUnshareable();

  return &(theString.value->data[charIndex]);
}
```

Much of this code is common to other `CharProxy` member functions, so I know you'd encapsulate it in a private member function that all would call.

A second difference between `char` s and the `CharProxy` s that stand for them becomes apparent if we have a template for reference-counted arrays that use proxy classes to distinguish lvalue and rvalue invocations of `operator[]` :

```
template<class T>                        // reference-counted array
class Array {                            // using proxies
public:
  class Proxy {
  public:
    Proxy(Array<T>& array, int index);
    Proxy& operator=(const T& rhs);
    operator T() const;
    ...
  };

  const Proxy operator[](int index) const;
  Proxy operator[](int index);
  ...
```

```
};
```

Consider how these arrays might be used:

```
Array<int> intArray;

...

intArray[5] = 22;                    // fine

intArray[5] += 5;                    // error!

++intArray[5];                       // error!
```

As expected, use of `operator[]` as the target of a simple assignment succeeds, but use of `operator[]` on the left-hand side of a call to `operator+=` or `operator++` fails. That's because `operator[]` returns a proxy, and there is no `operator+=` or `operator++` for `Proxy` objects. A similar situation exists for other operators that require lvalues, including `operator*=` , `operator<<=` , `operator--` , etc. If you want these operators to work with `operator[]` functions that return proxies, you must define each of these functions for the `Array<T>` ::`Proxy` class. That's a lot of work, and you probably don't want to do it. Unfortunately, you either do the work or you do without. Them's the breaks.

A related problem has to do with invoking member functions on real objects through proxies. To be blunt about it, you can't. For example, suppose we'd like to work with reference-counted arrays of rational numbers. We could define a class `Rational` and then use the `Array` template we just saw:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;
  ...
};

Array<Rational> array;
```

This is how we'd expect to be able to use such arrays, but, alas, we'd be disappointed:

```
cout << array[4].numerator();                    // error!

int denom = array[22].denominator();             // error!
```

By now the difficulty is predictable; `operator[]` returns a proxy for a rational number, not an actual `Rational` object. But the `numerator` and `denominator` member functions exist only for `Rational` s, not their proxies. Hence the complaints by your compilers. To make proxies behave like the objects they stand for, you must overload each function applicable to the real objects so it applies to proxies, too.

Yet another situation in which proxies fail to replace real objects is when being passed to functions that take references to non-const objects:

```
void swap(char& a, char& b);                    // swaps the value of a and b

String s = "+C+";                               // oops, should be "C++"

swap(s[0], s[1]);                               // this should fix the
                                                // problem, but it won't
                                                // compile
```

String::operator[] returns a CharProxy , but swap demands that its arguments be of type char& . A CharProxy may be implicitly converted into a char , but there is no conversion function to a char& . Furthermore, the char to which it may be converted can't be bound to swap 's char& parameters, because that char is a temporary object (it's operator char 's return value) and, as Item 19 explains, there are good reasons for refusing to bind temporary objects to non-const reference parameters.

A final way in which proxies fail to seamlessly replace real objects has to do with implicit type conversions. When a proxy object is implicitly converted into the real object it stands for, a user-defined conversion function is invoked. For instance, a CharProxy can be converted into the char it stands for by calling operator char . As Item 5 explains, compilers may use only one user-defined conversion function when converting a parameter at a call site into the type needed by the corresponding function parameter. As a result, it is possible for function calls that succeed when passed real objects to fail when passed proxies. For example, suppose we have a TVStation class and a function, watchTV :

```
class TVStation {
public:
  TVStation(int channel);
  ...

};

void watchTV(const TVStation& station, float hoursToWatch);
```

Thanks to implicit type conversion from int to TVStation (see Item 5), we could then do this:

```
watchTV(10, 2.5);                       // watch channel 10 for
                                        // 2.5 hours
```

Using the template for reference-counted arrays that use proxy classes to distinguish lvalue and rvalue invocations of operator[] , however, we could not do this:

```
Array<int> intArray;
```

```
intArray[4] = 10;

watchTV(intArray[4], 2.5);              // error! no conversion
                                        // from Proxy<int> to
                                        // TVStation
```

Given the problems that accompany implicit type conversions, it's hard to get too choked up about this. In fact, a better design for the `TVStation` class would declare its constructor `explicit` , in which case even the first call to `watchTV` would fail to compile. For all the details on implicit type conversions and how `explicit` affects them, see Item 5 .

Evaluation

Proxy classes allow you to achieve some types of behavior that are otherwise difficult or impossible to implement. Multidimensional arrays are one example, lvalue/rvalue differentiation is a second, suppression of implicit conversions (see Item 5) is a third.

At the same time, proxy classes have disadvantages. As function return values, proxy objects are temporaries (see Item 19), so they must be created and destroyed. That's not free, though the cost may be more than recouped through their ability to distinguish write operations from read operations. The very existence of proxy classes increases the complexity of software systems that employ them, because additional classes make things harder to design, implement, understand, and maintain, not easier.

Finally, shifting from a class that works with real objects to a class that works with proxies often changes the semantics of the class, because proxy objects usually exhibit behavior that is subtly different from that of the real objects they represent. Sometimes this makes proxies a poor choice when designing a system, but in many cases there is little need for the operations that would make the presence of proxies apparent to clients. For instance, few clients will want to take the address of an `Array1D` object in the two-dimensional array example we saw at the beginning of this Item, and there isn't much chance that an `ArrayIndex` object (see Item 5) would be passed to a function expecting a different type. In many cases, proxies can stand in for real objects perfectly acceptably. When they can, it is often the case that nothing else will do.

Back to Item 29: Reference counting
Continue to Item 31: Making functions virtual with respect to more than one object

Back to Item 30: Proxy classes
Continue to Miscellany

Item 31: Making functions virtual with respect to more than one object.

Sometimes, to borrow a phrase from Jacqueline Susann, once is not enough. Suppose, for example, you're bucking for one of those high-profile, high-prestige, high-paying programming jobs at that famous software company in Redmond, Washington — by which of course I mean Nintendo. To bring yourself to the attention of Nintendo's management, you might decide to write a video game. Such a game might take place in outer space and involve space ships, space stations, and asteroids.

As the ships, stations, and asteroids whiz around in your artificial world, they naturally run the risk of colliding with one another. Let's assume the rules for such collisions are as follows:

- If a ship and a station collide at low velocity, the ship docks at the station. Otherwise the ship and the station sustain damage that's proportional to the speed at which they collide.
- If a ship and a ship or a station and a station collide, both participants in the collision sustain damage that's proportional to the speed at which they hit.
- If a small asteroid collides with a ship or a station, the asteroid is destroyed. If it's a big asteroid, the ship or the station is destroyed.
- If an asteroid collides with another asteroid, both break into pieces and scatter little baby asteroids in all directions.

This may sound like a dull game, but it suffices for our purpose here, which is to consider how to structure the C++ code that handles collisions between objects.

We begin by noting that ships, stations, and asteroids share some common features. If nothing else, they're all in motion, so they all have a velocity that describes that motion. Given this commonality, it is natural to define a base class from which they all inherit. In practice, such a class is almost invariably an abstract base class, and, if you heed the warning I give in Item 33, base classes are always abstract. The hierarchy might therefore look like this:


```
class GameObject { ... };


class SpaceShip: public GameObject { ... };
```

```
class SpaceStation: public GameObject { ... };


class Asteroid: public GameObject { ... };
```

Now, suppose you're deep in the bowels of your program, writing the code to check for and handle object collisions. You might come up with a function that looks something like this:

```
void checkForCollision(GameObject& object1,
                       GameObject& object2)
{
  if (theyJustCollided(object1, object2)) {
    processCollision(object1, object2);
  }
  else {
    ...
  }
}
```

This is where the programming challenge becomes apparent. When you call `processCollision`, you know that `object1` and `object2` just collided, and you know that what happens in that collision depends on what `object1` really is and what `object2` really is, but you don't know what kinds of objects they really are; all you know is that they're both `GameObjects`. If the collision processing depended only on the dynamic type of `object1`, you could make `processCollision` virtual in `GameObject` and call `object1.processCollision(object2)`. You could do the same thing with `object2` if the details of the collision depended only on its dynamic type. What happens in the collision, however, depends on *both* their dynamic types. A function call that's virtual on only one object, you see, is not enough.

What you need is a kind of function whose behavior is somehow virtual on the types of more than one object. C++ offers no such function. Nevertheless, you still have to implement the behavior required above. The question, then, is how you are going to do it.

One possibility is to scrap the use of C++ and choose another programming language. You could turn to CLOS, for example, the Common Lisp Object System. CLOS supports what is possibly the most general object-oriented function-invocation mechanism one can imagine: *multi-methods*. A multi-method is a function that's virtual on as many parameters as you'd like, and CLOS goes even further by giving you substantial control over how calls to overloaded multi-methods are resolved.

Let us assume, however, that you must implement your game in C++ — that you must come up with your own way of implementing what is commonly referred to as *double-dispatching*. (The name comes from the object-oriented programming community, where what C++ programmers know as a virtual function call is termed a "message dispatch." A call that's virtual on two parameters is implemented through a "double dispatch." The generalization of this — a function

acting virtual on several parameters — is called *multiple dispatch*.) There are several approaches you might consider. None is without its disadvantages, but that shouldn't surprise you. C++ offers no direct support for double-dispatching, so you must yourself do the work compilers do when they implement virtual functions (see Item 24). If that were easy to do, we'd probably all be doing it ourselves and simply programming in C. We aren't and we don't, so fasten your seat belts, it's going to be a bumpy ride.

Using Virtual Functions and RTTI

Virtual functions implement a single dispatch; that's half of what we need; and compilers do virtual functions for us, so we begin by declaring a virtual function `collide` in `GameObject`. This function is overridden in the derived classes in the usual manner:

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  ...
};
```

Here I'm showing only the derived class `SpaceShip`, but `SpaceStation` and `Asteroid` are handled in exactly the same manner.

The most common approach to double-dispatching returns us to the unforgiving world of virtual function emulation via chains of `if-then-else`s. In this harsh world, we first discover the real type of `otherObject`, then we test it against all the possibilities:

```
// if we collide with an object of unknown type, we
// throw an exception of this type:
class CollisionWithUnknownObject {
public:
  CollisionWithUnknownObject(GameObject& whatWeHit);
  ...
};

void SpaceShip::collide(GameObject& otherObject)
{
  const type_info& objectType = typeid(otherObject);

  if (objectType == typeid(SpaceShip)) {
    SpaceShip& ss = static_cast<SpaceShip&>(otherObject);

    process a SpaceShip-SpaceShip collision;

  }
```

```
  else if (objectType == typeid(SpaceStation)) {
    SpaceStation& ss =
      static_cast<SpaceStation&>(otherObject);

    process a SpaceShip-SpaceStation collision;

  }

  else if (objectType == typeid(Asteroid)) {
    Asteroid& a = static_cast<Asteroid&>(otherObject);

    process a SpaceShip-Asteroid collision;

  }

  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

Notice how we need to determine the type of only one of the objects involved in the collision. The other object is `*this`, and its type is determined by the virtual function mechanism. We're inside a `SpaceShip` member function, so `*this` must be a `SpaceShip` object. Thus we only have to figure out the real type of `otherObject`.

There's nothing complicated about this code. It's easy to write. It's even easy to make work. That's one of the reasons RTTI is worrisome: it looks harmless. The true danger in this code is hinted at only by the final `else` clause and the exception that's thrown there.

We've pretty much bidden *adios* to encapsulation, because each `collide` function must be aware of each of its sibling classes, i.e., those classes that inherit from `GameObject`. In particular, if a new type of object — a new class — is added to the game, we must update each RTTI-based `if-then-else` chain in the program that might encounter the new object type. If we forget even a single one, the program will have a bug, and the bug will *not* be obvious. Furthermore, compilers are in no position to help us detect such an oversight, because they have no idea what we're doing (see also Item E39).

This kind of type-based programming has a long history in C, and one of the things we know about it is that it yields programs that are essentially unmaintainable. Enhancement of such programs eventually becomes unthinkable. This is the primary reason why virtual functions were invented in the first place: to shift the burden of generating and maintaining type-based function calls from programmers to compilers. When we employ RTTI to implement double-dispatching, we are harking back to the bad old days.

The techniques of the bad old days led to errors in C, and they'll lead to errors in C++, too. In recognition of our human frailty, we've included a final `else` clause in the `collide` function, a clause where control winds up if we hit an object we don't know about. Such a situation is, in principle, impossible, but where were our principles when we decided to use RTTI? There are various ways to handle such unanticipated interactions, but none is very satisfying. In this case,

we've chosen to throw an exception, but it's not clear how our callers can hope to handle the error any better than we can, since we've just run into something we didn't know existed.

Using Virtual Functions Only

There is a way to minimize the risks inherent in an RTTI approach to implementing double-dispatching, but before we look at that, it's convenient to see how to attack the problem using nothing but virtual functions. That strategy begins with the same basic structure as the RTTI approach. The `collide` function is declared virtual in `GameObject` and is redefined in each derived class. In addition, `collide` is overloaded in each class, one overloading for each derived class in the hierarchy:

```
class SpaceShip;                          // forward declarations
class SpaceStation;
class Asteroid;

class GameObject {
public:
  virtual void collide(GameObject&      otherObject) = 0;
  virtual void collide(SpaceShip&       otherObject) = 0;
  virtual void collide(SpaceStation&    otherObject) = 0;
  virtual void collide(Asteroid&        otherobject) = 0;
  ...

};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject&       otherObject);
  virtual void collide(SpaceShip&        otherObject);
  virtual void collide(SpaceStation&     otherObject);
  virtual void collide(Asteroid&         otherobject);
  ...

};
```

The basic idea is to implement double-dispatching as two single dispatches, i.e., as two separate virtual function calls: the first determines the dynamic type of the first object, the second determines that of the second object. As before, the first virtual call is to the `collide` function taking a `GameObject&` parameter. That function's implementation now becomes startlingly simple:

```
void SpaceShip::collide(GameObject& otherObject)
{
  otherObject.collide(*this);
}
```

At first glance, this appears to be nothing more than a recursive call to `collide` with the order of the parameters reversed, i.e., with `otherObject` becoming the object calling the member function

and `*this` becoming the function's parameter. Glance again, however, because this is *not* a recursive call. As you know, compilers figure out which of a set of functions to call on the basis of the static types of the arguments passed to the function. In this case, four different `collide` functions could be called, but the one chosen is based on the static type of `*this`. What is that static type? Being inside a member function of the class `SpaceShip`, `*this` must be of type `SpaceShip`. The call is therefore to the `collide` function taking a `SpaceShip&`, not the `collide` function taking a `GameObject&`.

All the `collide` functions are virtual, so the call inside `SpaceShip::collide` resolves to the implementation of `collide` corresponding to the real type of `otherObject`. Inside *that* implementation of `collide`, the real types of both objects are known, because the left-hand object is `*this` (and therefore has as its type the class implementing the member function) and the right-hand object's real type is `SpaceShip`, the same as the declared type of the parameter.

All this may be clearer when you see the implementations of the other `collide` functions in `SpaceShip`:

```
void SpaceShip::collide(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::collide(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::collide(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

As you can see, there's no muss, no fuss, no RTTI, no need to throw exceptions for unexpected object types. There can be no unexpected object types — that's the whole point of using virtual functions. In fact, were it not for its fatal flaw, this would be the perfect solution to the double-dispatching problem.

The flaw is one it shares with the RTTI approach we saw earlier: each class must know about its siblings. As new classes are added, the code must be updated. However, the *way* in which the code must be updated is different in this case. True, there are no `if-then-else`s to modify, but there is something that is often worse: each class definition must be amended to include a new virtual function. If, for example, you decide to add a new class `Satellite` (inheriting from `GameObject`) to your game, you'd have to add a new `collide` function to each of the existing classes in the program.

Modifying existing classes is something you are frequently in no position to do. If, instead of writing the entire video game yourself, you started with an off-the-shelf class library comprising a video game application framework, you might not have write access to the `GameObject` class or the framework classes derived from it. In that case, adding new member functions, virtual or otherwise,

is not an option. Alternatively, you may have *physical* access to the classes requiring modification, but you may not have *practical* access. For example, suppose you *were* hired by Nintendo and were put to work on programs using a library containing `GameObject` and other useful classes. Surely you wouldn't be the only one using that library, and Nintendo would probably be less than thrilled about recompiling every application using that library each time you decided to add a new type of object to your program. In practice, libraries in wide use are modified only rarely, because the cost of recompiling everything using those libraries is too great. (See Item E34 for information on how to design libraries that minimize compilation dependencies.)

The long and short of it is if you need to implement double-dispatching in your program, your best recourse is to modify your design to eliminate the need. Failing that, the virtual function approach is safer than the RTTI strategy, but it constrains the extensibility of your system to match that of your ability to edit header files. The RTTI approach, on the other hand, makes no recompilation demands, but, if implemented as shown above, it generally leads to software that is unmaintainable. You pays your money and you takes your chances.

Emulating Virtual Function Tables

There is a way to improve those chances. You may recall from Item 24 that compilers typically implement virtual functions by creating an array of function pointers (the vtbl) and then indexing into that array when a virtual function is called. Using a vtbl eliminates the need for compilers to perform chains of `if-then-else`-like computations, and it allows compilers to generate the same code at all virtual function call sites: determine the correct vtbl index, then call the function pointed to at that position in the vtbl.

There is no reason you can't do this yourself. If you do, you not only make your RTTI-based code more efficient (indexing into an array and following a function pointer is almost always more efficient than running through a series of `if-then-else` tests, and it generates less code, too), you also isolate the use of RTTI to a single location: the place where your array of function pointers is initialized. I should mention that the meek may inherit the earth, but the meek of heart may wish to take a few deep breaths before reading what follows.

We begin by making some modifications to the functions in the `GameObject` hierarchy:

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  virtual void hitSpaceShip(SpaceShip& otherObject);
  virtual void hitSpaceStation(SpaceStation& otherObject);
  virtual void hitAsteroid(Asteroid& otherobject);
  ...
};
```

```
void SpaceShip::hitSpaceShip(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::hitAsteroid(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

Like the RTTI-based hierarchy we started out with, the `GameObject` class contains only one function for processing collisions, the one that performs the first of the two necessary dispatches. Like the virtual-function-based hierarchy we saw later, each kind of interaction is encapsulated in a separate function, though in this case the functions have different names instead of sharing the name `collide`. There is a reason for this abandonment of overloading, and we shall see it soon. For the time being, note that the design above contains everything we need except an implementation for `SpaceShip::collide`; that's where the various `hit` functions will be invoked. As before, once we successfully implement the `SpaceShip` class, the `SpaceStation` and `Asteroid` classes will follow suit.

Inside `SpaceShip::collide`, we need a way to map the dynamic type of the parameter `otherObject` to a member function pointer that points to the appropriate collision-handling function. An easy way to do this is to create an associative array that, given a class name, yields the appropriate member function pointer. It's possible to implement `collide` using such an associative array directly, but it's a bit easier to understand what's going on if we add an intervening function, `lookup`, that takes a `GameObject` and returns the appropriate member function pointer. That is, you pass `lookup` a `GameObject`, and it returns a pointer to the member function to call when you collide with something of that `GameObject`'s type.

Here's the declaration of `lookup`:

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);


  static HitFunctionPtr lookup(const GameObject& whatWeHit);


  ...
};
```

The syntax of function pointers is never very pretty, and for member function pointers it's worse than usual, so we've typedefed HitFunctionPtr to be shorthand for a pointer to a member function of SpaceShip that takes a GameObject& and returns nothing.

Once we've got lookup, implementation of collide becomes the proverbial piece of cake:

```
void SpaceShip::collide(GameObject& otherObject)
{
  HitFunctionPtr hfp =
    lookup(otherObject);                // find the function to call

  if (hfp) {                            // if a function was found
    (this->*hfp)(otherObject);          // call it
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

Provided we've kept the contents of our associative array in sync with the class hierarchy under GameObject, lookup must always find a valid function pointer for the object we pass it. People are people, however, and mistakes have been known to creep into even the most carefully crafted software systems. That's why we still check to make sure a valid pointer was returned from lookup, and that's why we still throw an exception if the impossible occurs and the lookup fails.

All that remains now is the implementation of lookup. Given an associative array that maps from object types to member function pointers, the lookup itself is easy, but creating, initializing, and destroying the associative array is an interesting problem of its own.

Such an array should be created and initialized before it's used, and it should be destroyed when it's no longer needed. We could use new and delete to create and destroy the array manually, but that would be error-prone: how could we guarantee the array wasn't used before we got around to initializing it? A better solution is to have compilers automate the process, and we can do that by making the associative array static in lookup. That way it will be created and initialized the first time lookup is called, and it will be automatically destroyed sometime after main is exited (see Item E47).

Furthermore, we can use the map template from the Standard Template Library (see Item 35) as the associative array, because that's what a map is:

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  typedef map<string, HitFunctionPtr> HitMap;
  ...
};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
```

```
{
  static HitMap collisionMap;
  ...
}
```

Here, `collisionMap` is our associative array. It maps the name of a class (as a `string` object) to a `SpaceShip` member function pointer. Because `map<string, HitFunctionPtr>` is quite a mouthful, we use a typedef to make it easier to swallow. (For fun, try writing the declaration of `collisionMap` without using the `HitMap` and `HitFunctionPtr` typedefs. Most people will want to do this only once.)

Given `collisionMap`, the implementation of `lookup` is rather anticlimactic. That's because searching for something is an operation directly supported by the `map` class, and the one member function we can always (portably) call on the result of a `typeid` invocation is `name` (which, predictably[11], yields the name of the object's dynamic type). To implement `lookup`, then, we just find the entry in `collisionMap` corresponding to the dynamic type of `lookup`'s argument.

The code for `lookup` is straightforward, but if you're not familiar with the Standard Template Library (again, see Item 35), it may not seem that way. Don't worry. The comments in the function explain what's going on.

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;        // we'll see how to
                                     // initialize this below

  // look up the collision-processing function for the type
  // of whatWeHit. The value returned is a pointer-like
  // object called an "iterator" (see Item 35).
  HitMap::iterator mapEntry=
    collisionMap.find(typeid(whatWeHit).name());

  // mapEntry == collisionMap.end() if the lookup failed;
  // this is standard map behavior. Again, see Item 35.
  if (mapEntry == collisionMap.end()) return 0;

  // If we get here, the search succeeded. mapEntry
  // points to a complete map entry, which is a
  // (string, HitFunctionPtr) pair. We want only the
  // second part of the pair, so that's what we return.
  return (*mapEntry).second;
}
```

The final statement in the function returns `(*mapEntry).second` instead of the more conventional `mapEntry->second` in order to satisfy the vagaries of the STL. For details, see page 96.

Initializing Emulated Virtual Function Tables

Which brings us to the initialization of `collisionMap`. We'd like to say something like this,

```
// An incorrect implementation
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;

  collisionMap["SpaceShip"] = &hitSpaceShip;
  collisionMap["SpaceStation"] = &hitSpaceStation;
  collisionMap["Asteroid"] = &hitAsteroid;

  ...

}
```

but this inserts the member function pointers into collisionMap *each time* `lookup` is called, and that's needlessly inefficient. In addition, this won't compile, but that's a secondary problem we'll address shortly.

What we need now is a way to put the member function pointers into `collisionMap` only once — when `collisionMap` is created. That's easy enough to accomplish; we just write a private static member function called `initializeCollisionMap` to create and initialize our `map`, then we initialize `collisionMap` with `initializeCollisionMap`'s return value:

```
class SpaceShip: public GameObject {
private:
  static HitMap initializeCollisionMap();
  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap = initializeCollisionMap();
  ...
}
```

But this means we may have to pay the cost of copying the `map` object returned from `initializeCollisionMap` into `collisionMap` (see Items 19 and 20). We'd prefer not to do that. We wouldn't have to pay if `initializeCollisionMap` returned a pointer, but then we'd have to worry about making sure the `map` object the pointer pointed to was destroyed at an appropriate time.

Fortunately, there's a way for us to have it all. We can turn `collisionMap` into a smart pointer (see Item 28) that automatically deletes what it points to when the pointer itself is destroyed. In fact, the standard C++ library contains a template, `auto_ptr`, for just such a smart pointer (see Item 9). By making `collisionMap` a static `auto_ptr` in `lookup`, we can have `initializeCollisionMap` return a pointer to an initialized `map` object, yet never have to worry about a resource leak; the `map` to which `collisionMap` points will be automatically destroyed when `collisionMap` is. Thus:

```
class SpaceShip: public GameObject {
private:
  static HitMap * initializeCollisionMap();

  ...

};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static auto_ptr<HitMap>
    collisionMap(initializeCollisionMap());
  ...

}
```

The clearest way to implement `initializeCollisionMap` would seem to be this,

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

but as I noted earlier, this won't compile. That's because a `HitMap` is declared to hold pointers to member functions that all take the same type of argument, namely `GameObject`. But `hitSpaceShip` takes a `SpaceShip`, `hitSpaceStation` takes a `SpaceStation`, and, `hitAsteroid` takes an `Asteroid`. Even though `SpaceShip`, `SpaceStation`, and `Asteroid` can all be implicitly converted to `GameObject`, there is no such conversion for pointers to functions taking these argument types.

To placate your compilers, you might be tempted to employ `reinterpret_casts` (see Item 2), which are generally the casts of choice when converting between function pointer types:

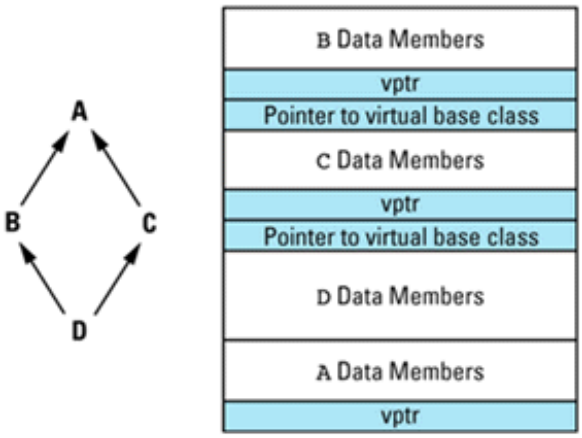
```
// A bad idea...
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceShip);

  (*phm)["SpaceStation"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceStation);
```

```
  (*phm)["Asteroid"] =
    reinterpret_cast<HitFunctionPtr>(&hitAsteroid);

  return phm;
}
```

This will compile, but it's a bad idea. It entails doing something you should never do: lying to your compilers. Telling them that `hitSpaceShip`, `hitSpaceStation`, and `hitAsteroid` are functions expecting a `GameObject` argument is simply not true. `hitSpaceShip` expects a `SpaceShip`, `hitSpaceStation` expects a `SpaceStation`, and `hitAsteroid` expects an `Asteroid`. The casts say otherwise. The casts lie.

More than morality is on the line here. Compilers don't like to be lied to, and they often find a way to exact revenge when they discover they've been deceived. In this case, they're likely to get back at you by generating bad code for functions you call through `*phm` in cases where `GameObject`'s derived classes employ multiple inheritance or have virtual base classes. In other words, if `SpaceStation`, `SpaceShip`, or `Asteroid` had other base classes (in addition to `GameObject`), you'd probably find that your calls to collision-processing functions in `collide` would behave quite rudely.

Consider again the A-B-C-D inheritance hierarchy and the possible object layout for a `D` object that is described in Item 24:



Each of the four class parts in a `D` object has a different address. This is important, because even though pointers and references behave differently (see Item 1), compilers typically *implement* references by using pointers in the generated code. Thus, pass-by-reference is typically

implemented by passing a pointer to an object. When an object with multiple base classes (such as a `D` object) is passed by reference, it is crucial that compilers pass the *correct* address — the one corresponding to the declared type of the parameter in the function being called.

But what if you've lied to your compilers and told them your function expects a `GameObject` when it really expects a `SpaceShip` or a `SpaceStation`? Then they'll pass the wrong address when you call the function, and the resulting runtime carnage will probably be gruesome. It will also be *very* difficult to determine the cause of the problem. There are good reasons why casting is discouraged. This is one of them.

Okay, so casting is out. Fine. But the type mismatch between the function pointers a `HitMap` is willing to contain and the pointers to the `hitSpaceShip`, `hitSpaceStation`, and `hitAsteroid` functions remains. There is only one way to resolve the conflict: change the types of the functions so they all take `GameObject` arguments:

```
class GameObject {                    // this is unchanged
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);

  // these functions now all take a GameObject parameter
  virtual void hitSpaceShip(GameObject& spaceShip);
  virtual void hitSpaceStation(GameObject& spaceStation);
  virtual void hitAsteroid(GameObject& asteroid);
  ...
};
```

Our solution to the double-dispatching problem that was based on virtual functions overloaded the function name `collide`. Now we are in a position to understand why we didn't follow suit here — why we decided to use an associative array of member function pointers instead. All the `hit` functions take the same parameter type, so we must give them different names.

Now we can write `initializeCollisionMap` the way we always wanted to:

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

Regrettably, our `hit` functions now get a general `GameObject` parameter instead of the derived class parameters they expect. To bring reality into accord with expectation, we must resort to a `dynamic_cast` (see Item 2) at the top of each function:

```
void SpaceShip::hitSpaceShip(GameObject& spaceShip)
{
  SpaceShip& otherShip=
    dynamic_cast<SpaceShip&>(spaceShip);

  process a SpaceShip-SpaceShip collision;

}

void SpaceShip::hitSpaceStation(GameObject& spaceStation)
{
  SpaceStation& station=
    dynamic_cast<SpaceStation&>(spaceStation);

  process a SpaceShip-SpaceStation collision;

}

void SpaceShip::hitAsteroid(GameObject& asteroid)
{
  Asteroid& theAsteroid =
    dynamic_cast<Asteroid&>(asteroid);

  process a SpaceShip-Asteroid collision;

}
```

Each of the `dynamic_casts` will throw a `bad_cast` exception if the cast fails. They should never fail, of course, because the `hit` functions should never be called with incorrect parameter types. Still, we're better off safe than sorry.

Using Non-Member Collision-Processing Functions

We now know how to build a vtbl-like associative array that lets us implement the second half of a double-dispatch, and we know how to encapsulate the details of the associative array inside a lookup function. Because this array contains pointers to *member* functions, however, we still have to modify class definitions if a new type of `GameObject` is added to the game, and that means everybody has to recompile, even people who don't care about the new type of object. For example, if `Satellite` were added to our game, we'd have to augment the `SpaceShip` class with a declaration of a function to handle collisions between satellites and spaceships. All `SpaceShip` clients would then have to recompile, even if they couldn't care less about the existence of satellites. This is the problem that led us to reject the implementation of double-dispatching based purely on virtual functions, and that solution was a lot less work than the one we've just seen.

The recompilation problem would go away if our associative array contained pointers to non-

member functions. Furthermore, switching to non-member collision-processing functions would let us address a design question we have so far ignored, namely, in which class should collisions between objects of different types be handled? With the implementation we just developed, if object 1 and object 2 collide and object 1 happens to be the left-hand argument to `processCollision`, the collision will be handled inside the class for object 1. If object 2 happens to be the left-hand argument to `processCollision`, however, the collision will be handled inside the class for object 2. Does this make sense? Wouldn't it be better to design things so that collisions between objects of types A and B are handled by neither A nor B but instead in some neutral location outside both classes?

If we move the collision-processing functions out of our classes, we can give clients header files that contain class definitions without any `hit` or `collide` functions. We can then structure our implementation file for `processCollision` as follows:

```
#include "SpaceShip.h"
#include "SpaceStation.h"
#include "Asteroid.h"

namespace {                    // unnamed namespace – see below

  // primary collision-processing functions
  void shipAsteroid(GameObject& spaceShip,
                    GameObject& asteroid);

  void shipStation(GameObject& spaceShip,
                   GameObject& spaceStation);

  void asteroidStation(GameObject& asteroid,
                       GameObject& spaceStation);
  ...

  // secondary collision-processing functions that just
  // implement symmetry: swap the parameters and call a
  // primary function
  void asteroidShip(GameObject& asteroid,
                    GameObject& spaceShip)
  { shipAsteroid(spaceShip, asteroid); }

  void stationShip(GameObject& spaceStation,
                   GameObject& spaceShip)
  { shipStation(spaceShip, spaceStation); }

  void stationAsteroid(GameObject& spaceStation,
                       GameObject& asteroid)
  { asteroidStation(asteroid, spaceStation); }

  ...

  // see below for a description of these types/functions
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  typedef map< pair<string,string>, HitFunctionPtr > HitMap;
```

```
  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2);

  HitMap * initializeCollisionMap();

  HitFunctionPtr lookup(const string& class1,
                        const string& class2);

} // end namespace

void processCollision(GameObject& object1,
                      GameObject& object2)
{
  HitFunctionPtr phf = lookup(typeid(object1).name(),
                              typeid(object2).name());

  if (phf) phf(object1, object2);
  else throw UnknownCollision(object1, object2);
}
```

Note the use of the unnamed namespace to contain the functions used to implement `processCollision`. Everything in such an unnamed namespace is private to the current translation unit (essentially the current file) — it's just like the functions were declared `static` at file scope. With the advent of namespaces, however, statics at file scope have been deprecated, so you should accustom yourself to using unnamed namespaces as soon as your compilers support them.

Conceptually, this implementation is the same as the one that used member functions, but there are some minor differences. First, `HitFunctionPtr` is now a typedef for a pointer to a non-member function. Second, the exception class `CollisionWithUnknownObject` has been renamed `UnknownCollision` and modified to take two objects instead of one. Finally, `lookup` must now take two type names and perform both parts of the double-dispatch. This means our collision map must now hold three pieces of information: two types names and a `HitFunctionPtr`.

As fate would have it, the standard `map` class is defined to hold only two pieces of information. We can finesse that problem by using the standard `pair` template, which lets us bundle the two type names together as a single object. `initializeCollisionMap`, along with its `makeStringPair` helper function, then looks like this:

```
// we use this function to create pair<string,string>
// objects from two char* literals. It's used in
// initializeCollisionMap below. Note how this function
// enables the return value optimization (see Item 20).

namespace {          // unnamed namespace again — see below

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2)
  { return pair<string,string>(s1, s2);   }

} // end namespace
```

```
namespace {          // still the unnamed namespace – see below

  HitMap * initializeCollisionMap()
  {
    HitMap *phm = new HitMap;

    (*phm)[makeStringPair("SpaceShip","Asteroid")] =
      &shipAsteroid;

    (*phm)[makeStringPair("SpaceShip", "SpaceStation")] =
      &shipStation;

    ...

    return phm;
  }

} // end namespace
```

`lookup` must also be modified to work with the `pair<string, string>` objects that now comprise the first component of the collision map:

```
namespace {          // I explain this below – trust me

  HitFunctionPtr lookup(const string& class1,
                        const string& class2)
  {
    static auto_ptr<HitMap>
      collisionMap(initializeCollisionMap());

    // see below for a description of make_pair
    HitMap::iterator mapEntry=
      collisionMap->find(make_pair(class1, class2));

    if (mapEntry == collisionMap->end()) return 0;

    return (*mapEntry).second;
  }

} // end namespace
```

This is almost exactly what we had before. The only real difference is the use of the `make_pair` function in this statement:

```
HitMap::iterator mapEntry=
  collisionMap->find(make_pair(class1, class2));
```

`make_pair` is just a convenience function (template) in the standard library (see Item E49 and Item 35) that saves us the trouble of specifying the types when constructing a `pair` object. We could just as well have written the statement like this:

```
HitMap::iterator mapEntry=
  collisionMap->find(pair<string,string>(class1, class2));
```

This calls for more typing, however, and specifying the types for the `pair` is redundant (they're the same as the types of `class1` and `class2`), so the `make_pair` form is more commonly used.

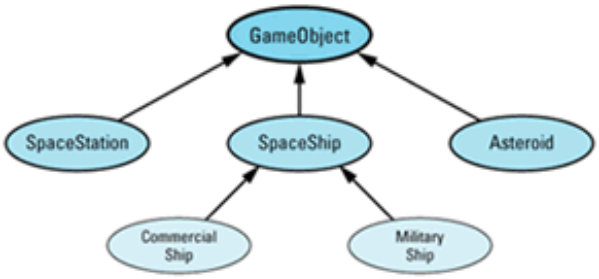
Because `makeStringPair`, `initializeCollisionMap`, and `lookup` were declared inside an unnamed namespace, each must be implemented within the same namespace. That's why the implementations of the functions above are in the unnamed namespace (for the same translation unit as their declarations): so the linker will correctly associate their definitions (i.e., their implementations) with their earlier declarations.

We have finally achieved our goals. If new subclasses of `GameObject` are added to our hierarchy, existing classes need not recompile (unless they wish to use the new classes). We have no tangle of RTTI-based `switch` or `if-then-else` conditionals to maintain. The addition of new classes to the hierarchy requires only well-defined and localized changes to our system: the addition of one or more map insertions in `initializeCollisionMap` and the declarations of the new collision-processing functions in the unnamed namespace associated with the implementation of `processCollision`. It may have been a lot of work to get here, but at least the trip was worthwhile. Yes? Yes?

Maybe.

Inheritance and Emulated Virtual Function Tables

There is one final problem we must confront. (If, at this point, you are wondering if there will *always* be one final problem to confront, you have truly come to appreciate the difficulty of designing an implementation mechanism for virtual functions.) Everything we've done will work fine as long as we never need to allow inheritance-based type conversions when calling collision-processing functions. But suppose we develop a game in which we must sometimes distinguish between commercial space ships and military space ships. We could modify our hierarchy as follows, where we've heeded the guidance of Item 33 and made the concrete classes `CommercialShip` and `MilitaryShip` inherit from the newly abstract class `SpaceShip`:



Suppose commercial and military ships behave identically when they collide with something. Then we'd expect to be able to use the same collision-processing functions we had before `CommercialShip` and `MilitaryShip` were added. In particular, if a `MilitaryShip` object and an `Asteroid` collided, we'd expect

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
```

to be called. It would not be. Instead, an `UnknownCollision` exception would be thrown. That's because `lookup` would be asked to find a function corresponding to the type names "MilitaryShip" and "Asteroid," and no such function would be found in `collisionMap`. Even though a `MilitaryShip` can be treated like a `SpaceShip`, `lookup` has no way of knowing that.

Furthermore, there is no easy way of telling it. If you need to implement double-dispatching and you need to support inheritance-based parameter conversions such as these, your only practical recourse is to fall back on the double-virtual-function-call mechanism we examined earlier. That implies you'll also have to put up with everybody recompiling when you add to your inheritance hierarchy, but that's just the way life is sometimes.

Initializing Emulated Virtual Function Tables (Reprise)

That's really all there is to say about double-dispatching, but it would be unpleasant to end the discussion on such a downbeat note, and unpleasantness is, well, unpleasant. Instead, let's conclude by outlining an alternative approach to initializing `collisionMap`.

As things stand now, our design is entirely static. Once we've registered a function for processing collisions between two types of objects, that's it; we're stuck with that function forever. What if we'd like to add, remove, or change collision-processing functions as the game proceeds? There's no way to do it.

But there can be. We can turn the concept of a map for storing collision-processing functions into a class that offers member functions allowing us to modify the contents of the map dynamically. For example:

```
class CollisionMap {
public:
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);

  void addEntry(const string& type1,
                const string& type2,
                HitFunctionPtr collisionFunction,
                bool symmetric = true);              // see below

  void removeEntry(const string& type1,
                   const string& type2);

  HitFunctionPtr lookup(const string& type1,
                        const string& type2);

  // this function returns a reference to the one and only
  // map — see Item 26
  static CollisionMap& theCollisionMap();

private:
  // these functions are private to prevent the creation
  // of multiple maps — see Item 26
  CollisionMap();
  CollisionMap(const CollisionMap&);
};
```

This class lets us add entries to the map, remove them from it, and look up the collision-processing function associated with a particular pair of type names. It also uses the techniques of Item 26 to limit the number of `CollisionMap` objects to one, because there is only one map in our system. (More complex games with multiple maps are easy to imagine.) Finally, it allows us to simplify the addition of symmetric collisions to the map (i.e., collisions in which the effect of an object of type T1 hitting an object of type T2 are the same as that of an object of type T2 hitting an object of type T1) by automatically adding the implied map entry when `addEntry` is called with the optional parameter `symmetric` set to `true`.

With the `CollisionMap` class, each client wishing to add an entry to the map does so directly:

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "Asteroid",
                                         &shipAsteroid);

void shipStation(GameObject& spaceShip,
                 GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "SpaceStation",
                                         &shipStation);

void asteroidStation(GameObject& asteroid,
```

```
                        GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("Asteroid",
                                         "SpaceStation",
                                         &asteroidStation);
...
```

Care must be taken to ensure that these map entries are added to the map before any collisions occur that would call the associated functions. One way to do this would be to have constructors in `GameObject` subclasses check to make sure the appropriate mappings had been added each time an object was created. Such an approach would exact a small performance penalty at runtime. An alternative would be to create a `RegisterCollisionFunction` class:

```
class RegisterCollisionFunction {
public:
  RegisterCollisionFunction(
          const string& type1,
          const string& type2,
          CollisionMap::HitFunctionPtr collisionFunction,
          bool symmetric = true)
  {
    CollisionMap::theCollisionMap().addEntry(type1, type2,
                                             collisionFunction,
                                             symmetric);
  }
};
```

Clients could then use global objects of this type to automatically register the functions they need:

```
RegisterCollisionFunction cf1("SpaceShip", "Asteroid",
                              &shipAsteroid);

RegisterCollisionFunction cf2("SpaceShip", "SpaceStation",
                              &shipStation);

RegisterCollisionFunction cf3("Asteroid", "SpaceStation",
                              &asteroidStation);

...

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

Because these objects are created before `main` is invoked, the functions their constructors register are also added to the map before `main` is called. If, later, a new derived class is added

```
class Satellite: public GameObject { ... };
```

and one or more new collision-processing functions are written,

```
void satelliteShip(GameObject& satellite,
                   GameObject& spaceShip);

void satelliteAsteroid(GameObject& satellite,
                       GameObject& asteroid);
```

these new functions can be similarly added to the map without disturbing existing code:

```
RegisterCollisionFunction cf4("Satellite", "SpaceShip",
                              &satelliteShip);

RegisterCollisionFunction cf5("Satellite", "Asteroid",
                              &satelliteAsteroid);
```

This doesn't change the fact that there's no perfect way to implement multiple dispatch, but it does make it easy to provide data for a map-based implementation if we decide such an approach is the best match for our needs.

Back to Item 30: Proxy classes
Continue to Miscellany

---

[11] It turns out that it's not so predictable after all. The C++ standard doesn't specify the return value of `type_info::name`, and different implementations behave differently. (Given a class `SpaceShip`, for example, one implementation's `type_info::name` returns "`class SpaceShip`".) A better design would identify a class by the address of its associated `type_info` object, because that is guaranteed to be unique. `HitMap` would then be declared to be of type `map<const type_info*, HitFunctionPtr>`.
Return

Back to Item 31: Making functions virtual with respect to more than one object
Continue to Item 32: Program in the future tense

Miscellany

We thus arrive at the organizational back of the bus, the chapter containing the guidelines no one else would have. We begin with two Items on C++ software development that describe how to design systems that accommodate change. One of the strengths of the object-oriented approach to systems building is its support for change, and these Items describe specific steps you can take to fortify your software against the slings and arrows of a world that refuses to stand still.

We then examine how to combine C and C++ in the same program. This necessarily leads to consideration of extralinguistic issues, but C++ exists in the real world, so sometimes we must confront such things.

Finally, I summarize changes to the ◦C++ language standard since publication of the *de facto* reference. I especially cover the sweeping changes that have been made in the standard library (see also Item E49). If you have not been following the standardization process closely, you are probably in for some surprises -- many of them quite pleasant.

Back to Item 31: Making functions virtual with respect to more than one object
Continue to Item 32: Program in the future tense

Item 32: Program in the future tense.

Things change.

As software developers, we may not know much, but we do know that things will change. We don't necessarily know what will change, how the changes will be brought about, when the changes will occur, or why they will take place, but we do know this: things will change.

Good software adapts well to change. It accommodates new features, it ports to new platforms, it adjusts to new demands, it handles new inputs. Software this flexible, this robust, and this reliable does not come about by accident. It is designed and implemented by programmers who conform to the constraints of today while keeping in mind the probable needs of tomorrow. This kind of software — software that accepts change gracefully — is written by people who *program in the future tense*.

To program in the future tense is to accept that things will change and to be prepared for it. It is to recognize that new functions will be added to libraries, that new overloadings will occur, and to watch for the potentially ambiguous function calls that might result (see Item E26). It is to acknowledge that new classes will be added to hierarchies, that present-day derived classes may be tomorrow's base classes, and to prepare for that possibility. It is to accept that new applications will be written, that functions will be called in new contexts, and to write those functions so they continue to perform correctly. It is to remember that the programmers charged with software maintenance are typically not the code's original developers, hence to design and implement in a fashion that facilitates comprehension, modification, and enhancement by others.

One way to do this is to express design constraints in C++ instead of (or in addition to) comments or other documentation. For example, if a class is designed to never have derived classes, don't just put a comment in the header file above the class, use C++ to prevent derivation; Item 26 shows you how. If a class requires that all instances be on the heap, don't just tell clients that, enforce the restriction by applying the approach of Item 27. If copying and assignment make no sense for a class, prevent those operations by declaring the copy constructor and the assignment operator private (see Item E27). C++ offers great power, flexibility, and expressiveness. Use these characteristics of the language to enforce the design decisions in your programs.

Given that things will change, write classes that can withstand the rough-and-tumble world of software evolution. Avoid "demand-paged" virtual functions, whereby you make no functions virtual unless somebody comes along and demands that you do it. Instead, determine the *meaning* of a function and whether it makes sense to let it be redefined in derived classes. If it does, declare it virtual, even if nobody redefines it right away. If it doesn't, declare it nonvirtual, and don't change

it later just because it would be convenient for someone; make sure the change makes sense in the context of the entire class and the abstraction it represents (see Item E36).

Handle assignment and copy construction in every class, even if "nobody ever does those things." Just because they don't do them now doesn't mean they won't do them in the future (see Item E18). If these functions are difficult to implement, declare them `private` (see Item E27). That way no one will inadvertently call compiler-generated functions that do the wrong thing (as often happens with default assignment operators and copy constructors — see Item E11).

Adhere to the principle of least astonishment: strive to provide classes whose operators and functions have a natural syntax and an intuitive semantics. Preserve consistency with the behavior of the built-in types: when in doubt, do as the `int`s do.

Recognize that anything somebody *can* do, they *will* do. They'll throw exceptions, they'll assign objects to themselves, they'll use objects before giving them values, they'll give objects values and never use them, they'll give them huge values, they'll give them tiny values, they'll give them null values. In general, if it will compile, somebody will do it. As a result, make your classes easy to use correctly and hard to use incorrectly. Accept that clients will make mistakes, and design your classes so you can prevent, detect, or correct such errors (see, for example, Item 33 and Item E46).

Strive for portable code. It's not much harder to write portable programs than to write unportable ones, and only rarely will the difference in performance be significant enough to justify unportable constructs (see Item 16). Even programs designed for custom hardware often end up being ported, because stock hardware generally achieves an equivalent level of performance within a few years. Writing portable code allows you to switch platforms easily, to enlarge your client base, and to brag about supporting open systems. It also makes it easier to recover if you bet wrong in the operating system sweepstakes.

Design your code so that when changes are necessary, the impact is localized. Encapsulate as much as you can; make implementation details private (e.g., Item E20). Where applicable, use unnamed namespaces or file-`static` objects and functions (see Item 31). Try to avoid designs that lead to virtual base classes, because such classes must be initialized by every class derived from them — even those derived indirectly (see Item 4 and Item E43). Avoid RTTI-based designs that make use of cascading `if-then-else` statements (see Item 31 again, then see Item E39 for good measure). Every time the class hierarchy changes, each set of statements must be updated, and if you forget one, you'll receive no warning from your compilers.

These are well known and oft-repeated exhortations, but most programmers are still stuck in the present tense. As are many authors, unfortunately. Consider this advice by a well-regarded C++ expert:

> You need a virtual destructor whenever someone deletes a `B*` that actually points to a `D`.

Here `B` is a base class and `D` is a derived class. In other words, this author suggests that if your program looks like this, you don't need a virtual destructor in `B`:

```
class B { ... };                    // no virtual dtor needed
class D: public B { ... };

B *pb = new D;
```

However, the situation changes if you add this statement:

```
delete pb;                          // NOW you need the virtual
                                    // destructor in B
```

The implication is that a minor change to client code — the addition of a `delete` statement — can result in the need to change the class definition for `B`. When that happens, all `B`'s clients must recompile. Following this author's advice, then, the addition of a single statement in one function can lead to extensive code recompilation and relinking for all clients of a library. This is anything but effective software design.

On the same topic, a different author writes:

> If a public base class does not have a virtual destructor, no derived class nor members of a derived class should have a destructor.

In other words, this is okay,

```
class string {                      // from the standard C++ library
public:
  ~string();
};

class B { ... };                    // no data members with dtors,
                                    // no virtual dtor needed
```

but if a new class is derived from `B`, things change:

```
class D: public B {
  string name;                      // NOW ~B needs to be virtual
```

```
};
```

Again, a small change to the way `B` is used (here, the addition of a derived class that contains a member with a destructor) may necessitate extensive recompilation and relinking by clients. But small changes in software should have small impacts on systems. This design fails that test.

The same author writes:

> If a multiple inheritance hierarchy has any destructors, every base class should have a virtual destructor.

In all these quotations, note the present-tense thinking. How do clients manipulate pointers *now*? What class members have destructors *now*? What classes in the hierarchy have destructors *now*?

Future-tense thinking is quite different. Instead of asking how a class is used now, it asks how the class is *designed* to be used. Future-tense thinking says, if a class is *designed* to be used as a base class (even if it's not used as one now), it should have a virtual destructor (see Item E14). Such classes behave correctly both now and in the future, and they don't affect other library clients when new classes derive from them. (At least, they have no effect as far as their destructor is concerned. If additional changes to the class are required, other clients may be affected.)

A commercial class library (one that predates the `string` specification in the C++ library standard) contains a string class with no virtual destructor. The vendor's explanation?

> We didn't make the destructor virtual, because we didn't want `String` to have a vtbl. We have no intention of ever having a `String*`, so this is not a problem. We are well aware of the difficulties this could cause.

Is this present-tense or future-tense thinking?

Certainly the vtbl issue is a legitimate technical concern (see Item 24 and Item E14). The implementation of most `String` classes contains only a single `char*` pointer inside each `String` object, so adding a vptr to each `String` would double the size of those objects. It is easy to understand why a vendor would be unwilling to do that, especially for a highly visible, heavily used class like `String`. The performance of such a class might easily fall within the 20% of a program that makes a difference (see Item 16).

Still, the total memory devoted to a string object — the memory for the object itself plus the heap memory needed to hold the string's value — is typically much greater than just the space needed to hold a `char*` pointer. From this perspective, the overhead imposed by a vptr is less significant.

Nevertheless, it is a legitimate technical consideration. (Certainly the ◦ISO/ANSI standardization committee seems to think so: the standard `string` type has a nonvirtual destructor.)

Somewhat more troubling is the vendor's remark, "We have no intention of ever having a `String*`, so this is not a problem." That may be true, but their `String` class is part of a library they make available to *thousands* of developers. That's a lot of developers, each with a different level of experience with C++, each doing something unique. Do those developers understand the consequences of there being no virtual destructor in `String`? Are they likely to know that because `String` has no virtual destructor, deriving new classes from `String` is a high-risk venture? Is this vendor confident their clients will understand that in the absence of a virtual destructor, deleting objects through `String*` pointers will not work properly and RTTI operations on pointers and references to `String`s may return incorrect information? Is this class easy to use correctly and hard to use incorrectly?

This vendor should provide documentation for its `String` class that makes clear the class is not designed for derivation, but what if programmers overlook the caveat or flat-out fail to read the documentation?

An alternative would be to use C++ itself to prohibit derivation. Item 26 describes how to do this by limiting object creation to the heap and then using `auto_ptr` objects to manipulate the heap objects. The interface for `String` creation would then be both unconventional and inconvenient, requiring this,

```
auto_ptr<String> ps(String::makeString("Future tense C++"));

...                                  // treat ps as a pointer to
                                     // a String object, but don't
                                     // worry about deleting it
```

instead of this,

```
String s("Future tense C++");
```

but perhaps the reduction in the risk of improperly behaving derived classes would be worth the syntactic inconvenience. (For `String`, this is unlikely to be the case, but for other classes, the trade-off might well be worth it.)

There is a need, of course, for present-tense thinking. The software you're developing has to work with current compilers; you can't afford to wait until the latest language features are implemented. It has to run on the hardware you currently support and it must do so under configurations your clients have available; you can't force your customers to upgrade their systems or modify their operating environment. It has to offer acceptable performance now; promises of smaller, faster

programs some years down the line don't generally warm the cockles of potential customers' hearts. And the software you're working on must be available "soon," which often means some time in the recent past. These are important constraints. You cannot ignore them.

Future-tense thinking simply adds a few additional considerations:

- Provide complete classes (see Item E18), even if some parts aren't currently used. When new demands are made on your classes, you're less likely to have to go back and modify them.
- Design your interfaces to facilitate common operations and prevent common errors (see Item E46). Make the classes easy to use correctly, hard to use incorrectly. For example, prohibit copying and assignment for classes where those operations make no sense (see Item E27). Prevent partial assignments (see Item 33).
- If there is no great penalty for generalizing your code, generalize it. For example, if you are writing an algorithm for tree traversal, consider generalizing it to handle any kind of directed acyclic graph.

Future tense thinking increases the reusability of the code you write, enhances its maintainability, makes it more robust, and facilitates graceful change in an environment where change is a certainty. It must be balanced against present-tense constraints. Too many programmers focus exclusively on current needs, however, and in doing so they sacrifice the long-term viability of the software they design and implement. Be different. Be a renegade. Program in the future tense.

Back to Miscellany
Continue to Item 33: Make non-leaf classes abstract

Back to Item 32: Program in the future tense
Continue to Item 34: Understand how to combine C++ and C in the same program

Item 33: Make non-leaf classes abstract.

Suppose you're working on a project whose software deals with animals. Within this software, most animals can be treated pretty much the same, but two kinds of animals — lizards and chickens — require special handling. That being the case, the obvious way to relate the classes for animals, lizards, and chickens is like this:

The `Animal` class embodies the features shared by all the creatures you deal with, and the `Lizard` and `Chicken` classes specialize `Animal` in ways appropriate for lizards and chickens, respectively.

Here's a sketch of the definitions for these classes:

```
class Animal {
public:
  Animal& operator=(const Animal& rhs);
  ...

};

class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...

};
```

Only the assignment operators are shown here, but that's more than enough to keep us busy for a while. Consider this code:

```
Lizard liz1;
Lizard liz2;
```

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;


...


*pAnimal1 = *pAnimal2;
```

There are two problems here. First, the assignment operator invoked on the last line is that of the `Animal` class, even though the objects involved are of type `Lizard`. As a result, only the `Animal` part of `liz1` will be modified. This is a *partial* assignment. After the assignment, `liz1`'s `Animal` members have the values they got from `liz2`, but `liz1`'s `Lizard` members remain unchanged.

The second problem is that real programmers write code like this. It's not uncommon to make assignments to objects through pointers, especially for experienced C programmers who have moved to C++. That being the case, we'd like to make the assignment behave in a more reasonable fashion. As Item 32 points out, our classes should be easy to use correctly and difficult to use incorrectly, and the classes in the hierarchy above are easy to use incorrectly.

One approach to the problem is to make the assignment operators virtual. If `Animal::operator=` were virtual, the assignment would invoke the `Lizard` assignment operator, which is certainly the correct one to call. However, look what happens if we declare the assignment operators virtual:

```
class Animal {
public:
  virtual Animal& operator=(const Animal& rhs);
  ...
};

class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  ...
};

class Chicken: public Animal {
public:
  virtual Chicken& operator=(const Animal& rhs);
  ...
};
```

Due to relatively recent changes to the language, we can customize the return value of the

assignment operators so that each returns a reference to the correct class, but the rules of C++ force us to declare identical *parameter* types for a virtual function in every class in which it is declared. That means the assignment operator for the `Lizard` and `Chicken` classes must be prepared to accept *any* kind of `Animal` object on the right-hand side of an assignment. That, in turn, means we have to confront the fact that code like the following is legal:

```
Lizard liz;
Chicken chick;

Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;                // assign a chicken to
                                      // a lizard!
```

This is a mixed-type assignment: a `Lizard` is on the left and a `Chicken` is on the right. Mixed-type assignments aren't usually a problem in C++, because the language's strong typing generally renders them illegal. By making `Animal`'s assignment operator virtual, however, we opened the door to such mixed-type operations.

This puts us in a difficult position. We'd like to allow same-type assignments through pointers, but we'd like to forbid mixed-type assignments through those same pointers. In other words, we want to allow this,

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;                // assign a lizard to a lizard
```

but we want to prohibit this:

```
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;

...

*pAnimal1 = *pAnimal2;                // assign a chicken to a lizard
```

Distinctions such as these can be made only at runtime, because sometimes assigning `*pAnimal2` to `*pAnimal1` is valid, sometimes it's not. We thus enter the murky world of type-based runtime errors. In particular, we need to signal an error inside `operator=` if we're faced with a mixed-type assignment, but if the types are the same, we want to perform the assignment in the usual fashion.

We can use a `dynamic_cast` (see Item 2) to implement this behavior. Here's how to do it for `Lizard`'s assignment operator:

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  // make sure rhs is really a lizard
  const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);

  proceed with a normal assignment of rhs_liz to *this;

}
```

This function assigns `rhs` to `*this` only if `rhs` is really a `Lizard`. If it's not, the function propagates the `bad_cast` exception that `dynamic_cast` throws when the cast fails. (Actually, the type of the exception is `std::bad_cast`, because the components of the standard library, including the exceptions thrown by the standard components, are in the namespace `std`. For an overview of the standard library, see Item E49 and Item 35.)

Even without worrying about exceptions, this function seems needlessly complicated and expensive — the `dynamic_cast` must consult a `type_info` structure; see Item 24 — in the common case where one `Lizard` object is assigned to another:

```
Lizard liz1, liz2;

...

liz1 = liz2;                          // no need to perform a
                                      // dynamic_cast: this
                                      // assignment must be valid
```

We can handle this case without paying for the complexity or cost of a `dynamic_cast` by adding to `Lizard` the conventional assignment operator:

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);

  Lizard& operator=(const Lizard& rhs);          // add this

  ...

};

Lizard liz1, liz2;

...

liz1 = liz2;                                  // calls operator= taking
```

```
                                                    // a const Lizard&

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;                             // calls operator= taking
                                                    // a const Animal&
```

In fact, given this latter `operator=`, it's simplicity itself to implement the former one in terms of it:

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  return operator=(dynamic_cast<const Lizard&>(rhs));
}
```

This function attempts to cast `rhs` to be a `Lizard`. If the cast succeeds, the normal class assignment operator is called. Otherwise, a `bad_cast` exception is thrown.

Frankly, all this business of checking types at runtime and using `dynamic_casts` makes me nervous. For one thing, some compilers still lack support for `dynamic_cast`, so code that uses it, though theoretically portable, is not necessarily portable in practice. More importantly, it requires that clients of `Lizard` and `Chicken` be prepared to catch `bad_cast` exceptions and do something sensible with them each time they perform an assignment. In my experience, there just aren't that many programmers who are willing to program that way. If they don't, it's not clear we've gained a whole lot over our original situation where we were trying to guard against partial assignments.

Given this rather unsatisfactory state of affairs regarding virtual assignment operators, it makes sense to regroup and try to find a way to prevent clients from making problematic assignments in the first place. If such assignments are rejected during compilation, we don't have to worry about them doing the wrong thing.

The easiest way to prevent such assignments is to make `operator=` private in `Animal`. That way, lizards can be assigned to lizards and chickens can be assigned to chickens, but partial and mixed-type assignments are forbidden:

```
class Animal {
private:
  Animal& operator=(const Animal& rhs);              // this is now
  ...                                                // private

};

class Lizard: public Animal {
public:
```

```
  Lizard& operator=(const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...

};

Lizard liz1, liz2;
...
liz1 = liz2;                                    // fine

Chicken chick1, chick2;
...
chick1 = chick2;                                // also fine

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2;                          // error! attempt to call
                                                // private Animal::operator=
```

Unfortunately, `Animal` is a concrete class, and this approach also makes assignments between `Animal` objects illegal:

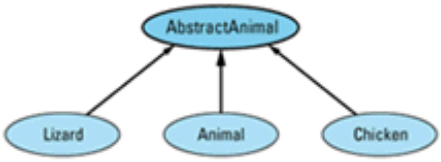
```
Animal animal1, animal2;

...

animal1 = animal2;                              // error! attempt to call
                                                // private Animal::operator=
```

Moreover, it makes it impossible to implement the `Lizard` and `Chicken` assignment operators correctly, because assignment operators in derived classes are responsible for calling assignment operators in their base classes (see Item E16):

```
Lizard& Lizard::operator=(const Lizard& rhs)
{
  if (this == &rhs) return *this;

  Animal::operator=(rhs);                       // error! attempt to call
                                                // private function. But
                                                // Lizard::operator= must
                                                // call this function to
  ...                                           // assign the Animal parts
}                                               // of *this!
```

We can solve this latter problem by declaring `Animal::operator= protected`, but the conundrum of allowing assignments between `Animal` objects while preventing partial assignments of `Lizard` and `Chicken` objects through `Animal` pointers remains. What's a poor programmer to do?

The easiest thing is to eliminate the need to allow assignments between `Animal` objects, and the easiest way to do that is to make `Animal` an abstract class. As an abstract class, `Animal` can't be instantiated, so there will be no need to allow assignments between `Animal`s. Of course, this leads to a new problem, because our original design for this system presupposed that `Animal` objects were necessary. There is an easy way around this difficulty. Instead of making `Animal` itself abstract, we create a new class — `AbstractAnimal`, say — consisting of the common features of `Animal`, `Lizard`, and `Chicken` objects, and we make *that* class abstract. Then we have each of our concrete classes inherit from `AbstractAnimal`. The revised hierarchy looks like this,



and the class definitions are as follows:

```
class AbstractAnimal {
protected:
  AbstractAnimal& operator=(const AbstractAnimal& rhs);

public:
  virtual ~AbstractAnimal() = 0;                    // see below
  ...

};

class Animal: public AbstractAnimal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};

class Lizard: public AbstractAnimal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};

class Chicken: public AbstractAnimal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
```

```
};
```

This design gives you everything you need. Homogeneous assignments are allowed for lizards, chickens, and animals; partial assignments and heterogeneous assignments are prohibited; and derived class assignment operators may call the assignment operator in the base class. Furthermore, none of the code written in terms of the `Animal`, `Lizard`, or `Chicken` classes requires modification, because these classes continue to exist and to behave as they did before `AbstractAnimal` was introduced. Sure, such code has to be recompiled, but that's a small price to pay for the security of knowing that assignments that compile will behave intuitively and assignments that would behave unintuitively won't compile.

For all this to work, `AbstractAnimal` must be abstract — it must contain at least one pure virtual function. In most cases, coming up with a suitable function is not a problem, but on rare occasions you may find yourself facing the need to create a class like `AbstractAnimal` in which none of the member functions would naturally be declared pure virtual. In such cases, the conventional technique is to make the destructor a pure virtual function; that's what's shown above. In order to support polymorphism through pointers correctly, base classes need virtual destructors anyway (see Item E14), so the only cost associated with making such destructors pure virtual is the inconvenience of having to implement them outside their class definitions. (For an example, see page 195.)

(If the notion of implementing a pure virtual function strikes you as odd, you just haven't been getting out enough. Declaring a function pure virtual doesn't mean it has no implementation, it means

- the current class is abstract, and
- any concrete class inheriting from the current class must declare the function as a "normal" virtual function (i.e., without the "`=0`").

True, most pure virtual functions are never implemented, but pure virtual destructors are a special case. They *must* be implemented, because they are called whenever a derived class destructor is invoked. Furthermore, they often perform useful tasks, such as releasing resources (see Item 9) or logging messages. Implementing pure virtual functions may be uncommon in general, but for pure virtual destructors, it's not just common, it's mandatory.)

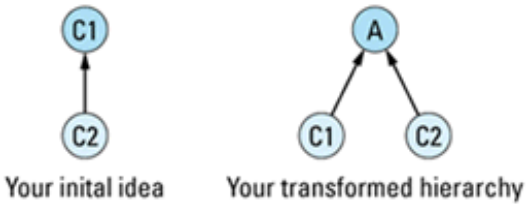
You may have noticed that this discussion of assignment through base class pointers is based on the assumption that concrete base classes like `Animal` contain data members. If there are no data members, you might point out, there is no problem, and it would be safe to have a concrete class inherit from a second, dataless, concrete class.

One of two situations applies to your data-free would-be concrete base class: either it might have data members in the future or it might not. If it might have data members in the future, all you're doing is postponing the problem until the data members are added, in which case you're merely

trading short-term convenience for long-term grief (see also Item 32). Alternatively, if the base class should truly never have any data members, that sounds very much like it should be an abstract class in the first place. What use is a concrete base class without data?

Replacement of a concrete base class like `Animal` with an abstract base class like `AbstractAnimal` yields benefits far beyond simply making the behavior of `operator=` easier to understand. It also reduces the chances that you'll try to treat arrays polymorphically, the unpleasant consequences of which are examined in Item 3. The most significant benefit of the technique, however, occurs at the design level, because replacing concrete base classes with abstract base classes forces you to explicitly recognize the existence of useful abstractions. That is, it makes you create new abstract classes for useful concepts, even if you aren't aware of the fact that the useful concepts exist.

If you have two concrete classes C1 and C2 and you'd like C2 to publicly inherit from C1, you should transform that two-class hierarchy into a three-class hierarchy by creating a new abstract class A and having both C1 and C2 publicly inherit from it:



The primary value of this transformation is that it forces you to identify the abstract class A. Clearly, C1 and C2 have something in common; that's why they're related by public inheritance (see Item E35). With this transformation, you must identify what that something is. Furthermore, you must formalize the something as a class in C++, at which point it becomes more than just a vague something, it achieves the status of a formal *abstraction*, one with well-defined member functions and well-defined semantics.

All of which leads to some worrisome thinking. After all, every class represents *some* kind of abstraction, so shouldn't we create two classes for every concept in our hierarchy, one being abstract (to embody the abstract part of the abstraction) and one being concrete (to embody the object-generation part of the abstraction)? No. If you do, you'll end up with a hierarchy with too many classes. Such a hierarchy is difficult to understand, hard to maintain, and expensive to compile. That is not the goal of object-oriented design.

The goal is to identify *useful* abstractions and to force them — and only them — into existence as abstract classes. But how do you identify useful abstractions? Who knows what abstractions might prove useful in the future? Who can predict who's going to want to inherit from what?

Well, I don't know how to predict the future uses of an inheritance hierarchy, but I do know one thing: the need for an abstraction in one context may be coincidental, but the need for an abstraction in more than one context is usually meaningful. Useful abstractions, then, are those that are needed in more than one context. That is, they correspond to classes that are useful in their own right (i.e., it is useful to have objects of that type) and that are also useful for purposes of one or more derived classes.

This is precisely why the transformation from concrete base class to abstract base class is useful: it forces the introduction of a new abstract class only when an existing concrete class is about to be used as a base class, i.e., when the class is about to be (re)used in a new context. Such abstractions are *useful*, because they have, through demonstrated need, shown themselves to be so.

The first time a concept is needed, we can't justify the creation of both an abstract class (for the concept) and a concrete class (for the objects corresponding to that concept), but the second time that concept is needed, we *can* justify the creation of both the abstract and the concrete classes. The transformation I've described simply mechanizes this process, and in so doing it forces designers and programmers to represent explicitly those abstractions that are useful, even if the designers and programmers are not consciously aware of the useful concepts. It also happens to make it a lot easier to bring sanity to the behavior of assignment operators.

Let's consider a brief example. Suppose you're working on an application that deals with moving information between computers on a network by breaking it into packets and transmitting them according to some protocol. All we'll consider here is the class or classes for representing packets. We'll assume such classes make sense for this application.

Suppose you deal with only a single kind of transfer protocol and only a single kind of packet. Perhaps you've heard that other protocols and packet types exist, but you've never supported them, nor do you have any plans to support them in the future. Should you make an abstract class for packets (for the concept that a packet represents) as well as a concrete class for the packets you'll actually be using? If you do, you could hope to add new packet types later without changing the base class for packets. That would save you from having to recompile packet-using applications if you add new packet types. But that design requires two classes, and right now you need only one (for the particular type of packets you use). Is it worth complicating your design now to allow for future extension that may never take place?

There is no unequivocally correct choice to be made here, but experience has shown it is nearly impossible to design good classes for concepts we do not understand well. If you create an abstract class for packets, how likely are you to get it right, especially since your experience is limited to only a single packet type? Remember that you gain the benefit of an abstract class for packets only if you can design that class so that future classes can inherit from it without its being changed in any way. (If it needs to be changed, you have to recompile all packet clients, and you've gained nothing.)

It is unlikely you could design a satisfactory abstract packet class unless you were well versed in many different kinds of packets and in the varied contexts in which they are used. Given your limited experience in this case, my advice would be not to define an abstract class for packets, adding one later only if you find a need to inherit from the concrete packet class.

The transformation I've described here is *a* way to identify the need for abstract classes, not *the* way. There are many other ways to identify good candidates for abstract classes; books on object-oriented analysis are filled with them. It's not the case that the only time you should introduce abstract classes is when you find yourself wanting to have a concrete class inherit from another concrete class. However, the desire to relate two concrete classes by public inheritance is usually indicative of a need for a new abstract class.

As is often the case in such matters, brash reality sometimes intrudes on the peaceful ruminations of theory. Third-party C++ class libraries are proliferating with gusto, and what are you to do if you find yourself wanting to create a concrete class that inherits from a concrete class in a library to which you have only read access?

You can't modify the library to insert a new abstract class, so your choices are both limited and unappealing:

- Derive your concrete class from the existing concrete class, and put up with the assignment-related problems we examined at the beginning of this Item. You'll also have to watch out for the array-related pitfalls described in Item 3.
- Try to find an abstract class higher in the library hierarchy that does most of what you need, then inherit from that class. Of course, there may not be a suitable class, and even if there is, you may have to duplicate a lot of effort that has already been put into the implementation of the concrete class whose functionality you'd like to extend.
- Implement your new class in terms of the library class you'd like to inherit from (see Items E40 and E42). For example, you could have an object of the library class as a data member, then reimplement the library class's interface in your new class:

```
class Window {                      // this is the library class
public:
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

  int width() const;
  int height() const;
};

class SpecialWindow {               // this is the class you
public:                             // wanted to have inherit
  ...                               // from Window

  // pass-through implementations of nonvirtual functions
  int width() const { return w.width(); }
  int height() const { return w.height(); }
```

```
  // new implementations of "inherited" virtual functions
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

private:
  Window w;
};
```

This strategy requires that you be prepared to update your class each time the library vendor updates the class on which you're dependent. It also requires that you be willing to forgo the ability to redefine virtual functions declared in the library class, because you can't redefine virtual functions unless you inherit them.

- Make do with what you've got. Use the concrete class that's in the library and modify your software so that the class suffices. Write non-member functions to provide the functionality you'd like to add to the class, but can't. The resulting software may not be as clear, as efficient, as maintainable, or as extensible as you'd like, but at least it will get the job done.

None of these choices is particularly attractive, so you have to apply some engineering judgment and choose the poison you find least unappealing. It's not much fun, but life's like that sometimes. To make things easier for yourself (and the rest of us) in the future, complain to the vendors of libraries whose designs you find wanting. With luck (and a lot of comments from clients), those designs will improve as time goes on.

Still, the general rule remains: non-leaf classes should be abstract. You may need to bend the rule when working with outside libraries, but in code over which you have control, adherence to it will yield dividends in the form of increased reliability, robustness, comprehensibility, and extensibility throughout your software.

Back to Item 32: Program in the future tense
Continue to Item 34: Understand how to combine C++ and C in the same program

Back to Item 33: Make non-leaf classes abstract
Continue to Item 35: Familiarize yourself with the language standard

Item 34: Understand how to combine C++ and C in the same program.

In many ways, the things you have to worry about when making a program out of some components in C++ and some in C are the same as those you have to worry about when cobbling together a C program out of object files produced by more than one C compiler. There is no way to combine such files unless the different compilers agree on implementation-dependent features like the size of `int`s and `double`s, the mechanism by which parameters are passed from caller to callee, and whether the caller or the callee orchestrates the passing. These pragmatic aspects of mixed-compiler software development are quite properly ignored by ◦language standardization efforts, so the only reliable way to know that object files from compiler A and compiler B can be safely combined in a program is to obtain assurances from the vendors of A and B that their products produce compatible output. This is as true for programs made up of C++ and C as it is for all-C++ or all-C programs, so before you try to mix C++ and C in the same program, make sure your C++ and C compilers generate compatible object files.

Having done that, there are four other things you need to consider: name mangling, initialization of statics, dynamic memory allocation, and data structure compatibility.

Name Mangling

Name mangling, as you may know, is the process through which your C++ compilers give each function in your program a unique name. In C, this process is unnecessary, because you can't overload function names, but nearly all C++ programs have at least a few functions with the same name. (Consider, for example, the iostream library, which declares several versions of `operator<<` and `operator>>`.) Overloading is incompatible with most linkers, because linkers generally take a dim view of multiple functions with the same name. Name mangling is a concession to the realities of linkers; in particular, to the fact that linkers usually insist on all function names being unique.

As long as you stay within the confines of C++, name mangling is not likely to concern you. If you have a function name `drawLine` that a compiler mangles into `xyzzy`, you'll always use the name `drawLine`, and you'll have little reason to care that the underlying object files happen to refer to `xyzzy`.

It's a different story if `drawLine` is in a C library. In that case, your C++ source file probably includes a header file that contains a declaration like this,

```
void drawLine(int x1, int y1, int x2, int y2);
```

and your code contains calls to `drawLine` in the usual fashion. Each such call is translated by your compilers into a call to the mangled name of that function, so when you write this,

```
drawLine(a, b, c, d);          // call to unmangled function name
```

your object files contain a function call that corresponds to this:

```
xyzzy(a, b, c, d);             // call to mangled function mame
```

But if `drawLine` is a C function, the object file (or archive or dynamically linked library, etc.) that contains the compiled version of `drawLine` contains a function called *`drawLine`*; no name mangling has taken place. When you try to link the object files comprising your program together, you'll get an error, because the linker is looking for a function called `xyzzy`, and there is no such function.

To solve this problem, you need a way to tell your C++ compilers not to mangle certain function names. You never want to mangle the names of functions written in other languages, whether they be in C, assembler, FORTRAN, Lisp, Forth, or what-have-you. (Yes, what-have-you would include COBOL, but then what would you have?) After all, if you call a C function named `drawLine`, it's really called `drawLine`, and your object code should contain a reference to that name, not to some mangled version of that name.

To suppress name mangling, use C++'s `extern "C"` directive:

```
// declare a function called drawLine; don't mangle
// its name
extern "C"
void drawLine(int x1, int y1, int x2, int y2);
```

Don't be drawn into the trap of assuming that where there's an `extern "C"`, there must be an `extern "Pascal"` and an `extern "FORTRAN"` as well. There's not, at least not in the standard. The best way to view `extern "C"` is not as an assertion that the associated function is written in C, but as a statement that the function should be called as if it *were* written in C. (Technically, `extern "C"` means the function has C linkage, but what that means is far from clear. One thing it always means, however, is that name mangling is suppressed.)

For example, if you were so unfortunate as to have to write a function in assembler, you could declare it `extern "C"`, too:

```
// this function is in assembler — don't mangle its name
```

```
extern "C" void twiddleBits(unsigned char bits);
```

You can even declare C++ functions `extern "C"`. This can be useful if you're writing a library in C++ that you'd like to provide to clients using other programming languages. By suppressing the name mangling of your C++ function names, your clients can use the natural and intuitive names you choose instead of the mangled names your compilers would otherwise generate:

```
// the following C++ function is designed for use outside
// C++ and should not have its name mangled
extern "C" void simulate(int iterations);
```

Often you'll have a slew of functions whose names you don't want mangled, and it would be a pain to precede each with `extern "C"`. Fortunately, you don't have to. `extern "C"` can also be made to apply to a whole set of functions. Just enclose them all in curly braces:

```
extern "C" {                            // disable name mangling for
                                        // all the following functions

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...
}
```

This use of `extern "C"` simplifies the maintenance of header files that must be used with both C++ and C. When compiling for C++, you'll want to include `extern "C"`, but when compiling for C, you won't. By taking advantage of the fact that the preprocessor symbol `__cplusplus` is defined only for C++ compilations, you can structure your polyglot header files as follows:

```
#ifdef __cplusplus

extern "C" {

#endif

  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...

#ifdef __cplusplus

}

#endif
```

There is, by the way, no such thing as a "standard" name mangling algorithm. Different compilers are free to mangle names in different ways, and different compilers do. This is a good thing. If all compilers mangled names the same way, you might be lulled into thinking they all generated compatible code. The way things are now, if you try to mix object code from incompatible C++ compilers, there's a good chance you'll get an error during linking, because the mangled names won't match up. This implies you'll probably have other compatibility problems, too, and it's better to find out about such incompatibilities sooner than later,

Initialization of Statics

Once you've mastered name mangling, you need to deal with the fact that in C++, lots of code can get executed before and after `main`. In particular, the constructors of static class objects and objects at global, namespace, and file scope are usually called before the body of `main` is executed. This process is known as *static initialization* (see Item E47). This is in direct opposition to the way we normally think about C++ and C programs, in which we view `main` as the entry point to execution of the program. Similarly, objects that are created through static initialization must have their destructors called during *static destruction*; that process typically takes place after `main` has finished executing.

To resolve the dilemma that `main` is supposed to be invoked first, yet objects need to be constructed before `main` is executed, many compilers insert a call to a special compiler-written function at the beginning of `main`, and it is this special function that takes care of static initialization. Similarly, compilers often insert a call to another special function at the end of `main` to take care of the destruction of static objects. Code generated for `main` often looks as if `main` had been written like this:

```
int main(int argc, char *argv[])
{
  performStaticInitialization();         // generated by the
                                         // implementation

  the statements you put in main go here;

  performStaticDestruction();            // generated by the
                                         // implementation
}
```

Now don't take this too literally. The functions `performStaticInitialization` and `performStaticDestruction` usually have much more cryptic names, and they may even be generated inline, in which case you won't see any functions for them in your object files. The important point is this: if a C++ compiler adopts this approach to the initialization and destruction of static objects, such objects will be neither initialized nor destroyed unless `main` is written in C++. Because this approach to static initialization and destruction is common, you should try to write `main` in C++ if you write any part of a software system in C++.

Sometimes it would seem to make more sense to write `main` in C — say if most of a program is in C and C++ is just a support library. Nevertheless, there's a good chance the C++ library contains static objects (if it doesn't now, it probably will in the future — see Item 32), so it's still a good idea to write `main` in C++ if you possibly can. That doesn't mean you need to rewrite your C code, however. Just rename the `main` you wrote in C to be `realMain`, then have the C++ version of `main` call `realMain`:

```
extern "C"                                  // implement this
int realMain(int argc, char *argv[]);       // function in C

int main(int argc, char *argv[])            // write this in C++
{
  return realMain(argc, argv);
}
```

If you do this, it's a good idea to put a comment above `main` explaining what is going on.

If you cannot write `main` in C++, you've got a problem, because there is no other portable way to ensure that constructors and destructors for static objects are called. This doesn't mean all is lost, it just means you'll have to work a little harder. Compiler vendors are well acquainted with this problem, so almost all provide some extralinguistic mechanism for initiating the process of static initialization and static destruction. For information on how this works with your compilers, dig into your compilers' documentation or contact their vendors.

Dynamic Memory Allocation

That brings us to dynamic memory allocation. The general rule is simple: the C++ parts of a program use `new` and `delete` (see Item 8), and the C parts of a program use `malloc` (and its variants) and `free`. As long as memory that came from `new` is deallocated via `delete` and memory that came from `malloc` is deallocated via `free`, all is well. Calling `free` on a `new`ed pointer yields undefined behavior, however, as does `delete`ing a `malloc`ed pointer. The only thing to remember, then, is to segregate rigorously your `new`s and `delete`s from your `malloc`s and `free`s.

Sometimes this is easier said than done. Consider the humble (but handy) `strdup` function, which, though standard in neither C nor C++, is nevertheless widely available:

```
char * strdup(const char *ps);          // return a copy of the
                                        // string pointed to by ps
```

If a memory leak is to be avoided, the memory allocated inside `strdup` must be deallocated by `strdup`'s caller. But how is the memory to be deallocated? By using `delete`? By calling `free`? If the

`strdup` you're calling is from a C library, it's the latter. If it was written for a C++ library, it's probably the former. What you need to do after calling `strdup`, then, varies not only from system to system, but also from compiler to compiler. To reduce such portability headaches, try to avoid calling functions that are neither in the standard library (see Item E49 and Item 35) nor available in a stable form on most computing platforms.

Data Structure Compatibility

Which brings us at long last to passing data between C++ and C programs. There's no hope of making C functions understand C++ features, so the level of discourse between the two languages must be limited to those concepts that C can express. Thus, it should be clear there's no portable way to pass objects or to pass pointers to member functions to routines written in C. C does understand normal pointers, however, so, provided your C++ and C compilers produce compatible output, functions in the two languages can safely exchange pointers to objects and pointers to non-member or static functions. Naturally, structs and variables of built-in types (e.g., `int`s, `char`s, etc.) can also freely cross the C++/C border.

Because the rules governing the layout of a `struct` in C++ are consistent with those of C, it is safe to assume that a structure definition that compiles in both languages is laid out the same way by both compilers. Such structs can be safely passed back and forth between C++ and C. If you add *nonvirtual* functions to the C++ version of the struct, its memory layout should not change, so objects of a struct (or class) containing only non-virtual functions should be compatible with their C brethren whose structure definition lacks only the member function declarations. Adding *virtual* functions ends the game, because the addition of virtual functions to a class causes objects of that type to use a different memory layout (see Item 24). Having a struct inherit from another struct (or class) usually changes its layout, too, so structs with base structs (or classes) are also poor candidates for exchange with C functions.

From a data structure perspective, it boils down to this: it is safe to pass data structures from C++ to C and from C to C++ provided the definition of those structures compiles in both C++ and C. Adding nonvirtual member functions to the C++ version of a struct that's otherwise compatible with C will probably not affect its compatibility, but almost any other change to the struct will.

Summary

If you want to mix C++ and C in the same program, remember the following simple guidelines:

- Make sure the C++ and C compilers produce compatible object files.
- Declare functions to be used by both languages `extern "C"`.
- If at all possible, write `main` in C++.
- Always use `delete` with memory from `new`; always use `free` with memory from `malloc`.
- Limit what you pass between the two languages to data structures that compile under C; the C++ version of structs may contain non-virtual member functions.

Back to Item 33: Make non-leaf classes abstract
Continue to Item 35: Familiarize yourself with the language standard

Back to Item 34: Understand how to combine C++ and C in the same program
Continue to Recommended Reading

Item 35: Familiarize yourself with the language standard.

Since its publication in 1990, *The Annotated C++ Reference Manual* (see page 285) has been the definitive reference for working programmers needing to know what is in C++ and what is not. In the years since the ARM (as it's fondly known) came out, the ISO/ANSI committee standardizing the language has changed (primarily extended) the language in ways both big and small. As a definitive reference, the ARM no longer suffices.

The post-ARM changes to C++ significantly affect how good programs are written. As a result, it is important for C++ programmers to be familiar with the primary ways in which the C++ specified by the standard differs from that described by the ARM.

The ISO/ANSI standard for C++ is what vendors will consult when implementing compilers, what authors will examine when preparing books, and what programmers will look to for definitive answers to questions about C++. Among the most important changes to C++ since the ARM are the following:

- **New features have been added**: RTTI, namespaces, `bool`, the `mutable` and `explicit` keywords, the ability to overload operators for enums, and the ability to initialize constant integral static class members within a class definition.
- **Templates have been extended**: member templates are now allowed, there is a standard syntax for forcing template instantiations, non-type arguments are now allowed in function templates, and class templates may themselves be used as template arguments.
- **Exception handling has been refined**: exception specifications are now more rigorously checked during compilation, and the `unexpected` function may now throw a `bad_exception` object.
- **Memory allocation routines have been modified**: `operator new[]` and `operator delete[]` have been added, the operators `new`/`new[]` now throw an exception if memory can't be allocated, and there are now alternative versions of the operators `new`/`new[]` that return 0 when an allocation fails (see Item E7).
- **New casting forms have been added**: `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`.
- **Language rules have been refined**: redefinitions of virtual functions need no longer have a return type that exactly matches that of the function they redefine, and the lifetime of temporary objects has been defined precisely.

Almost all these changes are described in *The Design and Evolution of C++* (see page 285). Current C++ textbooks (those written after 1994) should include them, too. (If you find one that doesn't, reject it.) In addition, *More Effective C++* (that's this book) contains examples of how to use most of these new features. If you're curious about something on this list, try looking it up in the

index.

The changes to C++ proper pale in comparison to what's happened to the standard library. Furthermore, the evolution of the standard library has not been as well publicized as that of the language. *The Design and Evolution of C++*, for example, makes almost no mention of the standard library. The books that do discuss the library are sometimes out of date, because the library changed quite substantially in 1994.

The capabilities of the standard library can be broken down into the following general categories (see also Item E49):

- **Support for the standard C library.** Fear not, C++ still remembers its roots. Some minor tweaks have brought the C++ version of the C library into conformance with C++'s stricter type checking, but for all intents and purposes, everything you know and love (or hate) about the C library continues to be knowable and lovable (or hateable) in C++, too.
- **Support for strings.** As Chair of the working group for the standard C++ library, Mike Vilot was told, "If there isn't a standard `string` type, there will be blood in the streets!" (Some people get so emotional.) Calm yourself and put away those hatchets and truncheons — the standard C++ library has strings.
- **Support for localization.** Different cultures use different character sets and follow different conventions when displaying dates and times, sorting strings, printing monetary values, etc. Localization support within the standard library facilitates the development of programs that accommodate such cultural differences.
- **Support for I/O.** The iostream library remains part of the C++ standard, but the committee has tinkered with it a bit. Though some classes have been eliminated (notably `iostream` and `fstream`) and some have been replaced (e.g., `string`-based `stringstreams` replace `char*`-based `strstreams`, which are now deprecated), the basic capabilities of the standard iostream classes mirror those of the implementations that have existed for several years.
- **Support for numeric applications.** Complex numbers, long a mainstay of examples in C++ texts, have finally been enshrined in the standard library. In addition, the library contains special array classes (`valarrays`) that restrict aliasing. These arrays are eligible for more aggressive optimization than are built-in arrays, especially on multiprocessing architectures. The library also provides a few commonly useful numeric functions, including partial sum and adjacent difference.
- **Support for general-purpose containers and algorithms.** Contained within the standard C++ library is a set of class and function templates collectively known as the Standard Template Library (STL). The STL is the most revolutionary part of the standard C++ library. I summarize its features below.

Before I describe the STL, though, I must dispense with two idiosyncrasies of the standard C++ library you need to know about.

First, almost everything in the library is a *template*. In this book, I may have referred to the standard

`string` class, but in fact there is no such class. Instead, there is a class template called `basic_string` that represents sequences of characters, and this template takes as a parameter the type of the characters making up the sequences. This allows for strings to be made up of `chars`, wide chars, Unicode chars, whatever.

What we normally think of as the `string` class is really the template instantiation `basic_string<char>`. Because its use is so common, the standard library provides a typedef:

```
typedef basic_string<char> string;
```

Even this glosses over many details, because the `basic_string` template takes three arguments; all but the first have default values. To *really* understand the `string` type, you must face this full, unexpurgated declaration of `basic_string`:

```
template<class charT,
         class traits = string_char_traits<charT>,
         class Allocator = allocator>
  class basic_string;
```

You don't need to understand this gobbledygook to use the `string` type, because even though `string` is a typedef for The Template Instantiation from Hell, it behaves as if it were the unassuming non-template class the typedef makes it appear to be. Just tuck away in the back of your mind the fact that if you ever need to customize the types of characters that go into strings, or if you want to fine-tune the behavior of those characters, or if you want to seize control over the way memory for strings is allocated, the `basic_string` template allows you to do these things.

The approach taken in the design of the `string` type — generalize it and make the generalization a template — is repeated throughout the standard C++ library. IOstreams? They're templates; a type parameter defines the type of character making up the streams. Complex numbers? Also templates; a type parameter defines how the components of the numbers should be stored. Valarrays? Templates; a type parameter specifies what's in each array. And of course the STL consists almost entirely of templates. If you are not comfortable with templates, now would be an excellent time to start making serious headway toward that goal.

The other thing to know about the standard library is that virtually everything it contains is inside the namespace `std`. To use things in the standard library without explicitly qualifying their names, you'll have to employ a `using` directive or (preferably) `using` declarations (see Item E28). Fortunately, this syntactic administrivia is automatically taken care of when you `#include` the appropriate headers.

The Standard Template Library

The biggest news in the standard C++ library is the STL, the Standard Template Library. (Since almost everything in the C++ library is a template, the name STL is not particularly descriptive. Nevertheless, this is the name of the containers and algorithms portion of the library, so good name or bad, this is what we use.)

The STL is likely to influence the organization of many — perhaps most — C++ libraries, so it's important that you be familiar with its general principles. They are not difficult to understand. The STL is based on three fundamental concepts: containers, iterators, and algorithms. Containers hold collections of objects. Iterators are pointer-like objects that let you walk through STL containers just as you'd use pointers to walk through built-in arrays. Algorithms are functions that work on STL containers and that use iterators to help them do their work.

It is easiest to understand the STL view of the world if we remind ourselves of the C++ (and C) rules for arrays. There is really only one rule we need to know: a pointer to an array can legitimately point to any element of the array *or to one element beyond the end of the array*. If the pointer points to the element beyond the end of the array, it can be compared only to other pointers to the array; the results of dereferencing it are undefined.

We can take advantage of this rule to write a function to find a particular value in an array. For an array of integers, our function might look like this:

```
int * find(int *begin, int *end, int value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

This function looks for `value` in the range between `begin` and `end` (excluding `end` — `end` points to one beyond the end of the array) and returns a pointer to the first occurrence of `value` in the array; if none is found, it returns `end`.

Returning `end` seems like a funny way to signal a fruitless search. Wouldn't 0 (the null pointer) be better? Certainly null seems more natural, but that doesn't make it "better." The `find` function must return some distinctive pointer value to indicate the search failed, and for this purpose, the `end` pointer is as good as the null pointer. In addition, as we'll soon see, the `end` pointer generalizes to other types of containers better than the null pointer.

Frankly, this is probably not the way you'd write the `find` function, but it's not unreasonable, and it generalizes astonishingly well. If you followed this simple example, you have mastered most of the ideas on which the STL is founded.

You could use the `find` function like this:

```
int values[50];

...

int *firstFive = find(values,        // search the range
                      values+50,     // values[0] - values[49]
                      5);            // for the value 5

if (firstFive != values+50) {        // did the search succeed?

  ...                                // yes

}
else {
  ...                                // no, the search failed
}
```

You can also use `find` to search subranges of the array:

```
int *firstFive = find(values,        // search the range
                      values+10,     // values[0] - values[9]
                      5);            // for the value 5

int age = 36;

...

int *firstValue = find(values+10,    // search the range
                       values+20,    // values[10] - values[19]
                       age);         // for the value in age
```

There's nothing inherent in the `find` function that limits its applicability to arrays of `int`s, so it should really be a template:

```
template<class T>
T * find(T *begin, T *end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

In the transformation to a template, notice how we switched from pass-by-value for `value` to pass-by-reference-to-`const`. That's because now that we're passing arbitrary types around, we have to worry about the cost of pass-by-value. Each by-value parameter costs us a call to the parameter's

constructor and destructor every time the function is invoked. We avoid these costs by using pass-by-reference, which involves no object construction or destruction (see Item E22).

This template is nice, but it can be generalized further. Look at the operations on `begin` and `end`. The only ones used are comparison for inequality, dereferencing, prefix increment (see Item 6), and copying (for the function's return value — see Item 19). These are all operations we can overload, so why limit `find` to using pointers? Why not allow any object that supports these operations to be used in addition to pointers? Doing so would free the `find` function from the built-in meaning of pointer operations. For example, we could define a pointer-like object for a linked list whose prefix increment operator moved us to the next element in the list.

This is the concept behind STL *iterators*. Iterators are pointer-like objects designed for use with STL containers. They are first cousins to the smart pointers of Item 28, but smart pointers tend to be more ambitious in what they do than do STL iterators. From a technical viewpoint, however, they are implemented using the same techniques.

Embracing the notion of iterators as pointer-like objects, we can replace the pointers in `find` with iterators, thus rewriting `find` like this:

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

Congratulations! You have just written part of the Standard Template Library. The STL contains dozens of algorithms that work with containers and iterators, and `find` is one of them.

Containers in STL include `bitset`, `vector`, `list`, `deque`, `queue`, `priority_queue`, `stack`, `set`, and `map`, and you can apply `find` to *any* of these container types:

```
list<char> charList;                 // create STL list object
                                     // for holding chars
...

// find the first occurrence of 'x' in charList
list<char>::iterator it = find(charList.begin(),
                               charList.end(),
                               'x');
```

"Whoa!", I hear you cry, "This doesn't look *anything* like it did in the array examples above!" Ah, but it does; you just have to know what to look for.

To call `find` for a `list` object, you need to come up with iterators that point to the first element of the list and to one past the last element of the list. Without some help from the `list` class, this is a difficult task, because you have no idea how a `list` is implemented. Fortunately, `list` (like all STL containers) obliges by providing the member functions `begin` and `end`. These member functions return the iterators you need, and it is those iterators that are passed into the first two parameters of `find` above.

When `find` is finished, it returns an iterator object that points to the found element (if there is one) or to `charList.end()` (if there's not). Because you know nothing about how `list` is implemented, you also know nothing about how iterators into `lists` are implemented. How, then, are you to know what type of object is returned by `find`? Again, the `list` class, like all STL containers, comes to the rescue: it provides a typedef, `iterator`, that is the type of iterators into `lists`. Since `charList` is a `list` of `chars`, the type of an iterator into such a list is `list<char>::iterator`, and that's what's used in the example above. (Each STL container class actually defines two iterator types, `iterator` and `const_iterator`. The former acts like a normal pointer, the latter like a pointer-to-`const`.)

Exactly the same approach can be used with the other STL containers. Furthermore, C++ pointers *are* STL iterators, so the original array examples work with the STL `find` function, too:

```
int values[50];

...

int *firstFive = find(values, values+50, 5);       // fine, calls
                                                   // STL find
```

At its core, STL is very simple. It is just a collection of class and function templates that adhere to a set of conventions. The STL collection classes provide functions like `begin` and `end` that return iterator objects of types defined by the classes. The STL algorithm functions move through collections of objects by using iterator objects over STL collections. STL iterators act like pointers. That's really all there is to it. There's no big inheritance hierarchy, no virtual functions, none of that stuff. Just some class and function templates and a set of conventions to which they all subscribe.

Which leads to another revelation: STL is extensible. You can add your own collections, algorithms, and iterators to the STL family. As long as you follow the STL conventions, the standard STL collections will work with your algorithms and your collections will work with the standard STL algorithms. Of course, your templates won't be part of the standard C++ library, but they'll be built on the same principles and will be just as reusable.

There is much more to the C++ library than I've described here. Before you can use the library effectively, you must learn more about it than I've had room to summarize, and before you can

write your own STL-compliant templates, you must learn more about the conventions of the STL. The standard C++ library is far richer than the C library, and the time you take to familiarize yourself with it is time well spent (see also Item E49). Furthermore, the design principles embodied by the library — those of generality, extensibility, customizability, efficiency, and reusability — are well worth learning in their own right. By studying the standard C++ library, you not only increase your knowledge of the ready-made components available for use in your software, you learn how to apply the features of C++ more effectively, and you gain insight into how to design better libraries of your own.

Back to Item 34: Understand how to combine C++ and C in the same program
Continue to Recommended Reading

Back to Item 35: : Familiarize yourself with the language standard
Continue to An `auto_ptr` Implementation

Recommended Reading

So your appetite for information on C++ remains unsated. Fear not, there's more — much more. In the sections that follow, I put forth my recommendations for further reading on C++. It goes without saying that such recommendations are both subjective and selective, but in view of the litigious age in which we live, it's probably a good idea to say it anyway.

Books

There are hundreds — possibly thousands — of books on C++, and new contenders join the fray with great frequency. I haven't seen all these books, much less read them, but my experience has been that while some books are very good, some of them, well, some of them aren't.

What follows is the list of books I find myself consulting when I have questions about software development in C++. Other good books are available, I'm sure, but these are the ones I use, the ones I can truly *recommend*.

A good place to begin is with the books that describe the language itself. Unless you are crucially dependent on the nuances of the official standards documents, I suggest you do, too.

- *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990, ISBN 0-201-51459-1.

- *The Design and Evolution of C++*, Bjarne Stroustrup, Addison-Wesley, 1994, ISBN 0-201-54330-3.

These books contain not just a description of what's in the language, they also explain the rationale behind the design decisions — something you won't find in the official standard documents. *The Annotated C++ Reference Manual* is now incomplete (several language features have been added since it was published — see Item 35) and is in some cases out of date, but it is still the best reference for the core parts of the language, including templates and exceptions. *The Design and Evolution of C++* covers most of what's missing in *The Annotated C++ Reference Manual*; the only thing it lacks is a discussion of the Standard Template Library (again, see Item 35). These books are not tutorials, they're references, but you can't truly understand C++ unless you understand the material in these books.

For a more general reference on the language, the standard library, and how to apply it, there is no better place to look than the book by the man responsible for C++ in the first place:

- *The C++ Programming Language (Third Edition)*, Bjarne Stroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4.

Stroustrup has been intimately involved in the language's design, implementation, application, and

standardization since its inception, and he probably knows more about it than anybody else does. His descriptions of language features make for dense reading, but that's primarily because they contain so much information. The chapters on the standard C++ library provide a good introduction to this crucial aspect of modern C++.

If you're ready to move beyond the language itself and are interested in how to apply it effectively, you might consider my other book on the subject:

- *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1998, ISBN 0-201-92488-9.

That book is organized similarly to this one, but it covers different (arguably more fundamental) material.

A book pitched at roughly the same level as my *Effective C++* books, but covering different topics, is

- *C++ Strategies and Tactics*, Robert Murray, Addison-Wesley, 1993, ISBN 0-201-56382-7.

Murray's book is especially strong on the fundamentals of template design, a topic to which he devotes two chapters. He also includes a chapter on the important topic of migrating from C development to C++ development. Much of my discussion on reference counting (see Item 29) is based on the ideas in *C++ Strategies and Tactics.*

If you're the kind of person who likes to learn proper programming technique by reading *code*, the book for you is

- *C++ Programming Style*, Tom Cargill, Addison-Wesley, 1992, ISBN 0-201-56365-7.

Each chapter in this book starts with some C++ software that has been published as an example of how to do something correctly. Cargill then proceeds to dissect — nay, *vivisect* — each program, identifying likely trouble spots, poor design choices, brittle implementation decisions, and things that are just plain wrong. He then iteratively rewrites each example to eliminate the weaknesses, and by the time he's done, he's produced code that is more robust, more maintainable, more efficient, and more portable, and it still fulfills the original problem specification. Anybody programming in C++ would do well to heed the lessons of this book, but it is especially important for those involved in code inspections.

One topic Cargill does not discuss in *C++ Programming Style* is exceptions. He turns his critical eye to this language feature in the following article, however, which demonstrates why writing exception-safe code is more difficult than most programmers realize:

- "Exception Handling: A False Sense of Security," *C++ Report*, Volume 6, Number 9, November-December 1994, pages 21-24.

If you are contemplating the use of exceptions, read this article before you proceed.

Once you've mastered the basics of C++ and are ready to start pushing the envelope, you must familiarize yourself with

- *Advanced C++: Programming Styles and Idioms*, James Coplien, Addison-Wesley, 1992, ISBN 0-201-54855-0.

I generally refer to this as "the LSD book," because it's purple and it will expand your mind. Coplien covers some straightforward material, but his focus is really on showing you how to do things in C++ you're not supposed to be able to do. You want to construct objects on top of one another? He shows you how. You want to bypass strong typing? He gives you a way. You want to add data and functions to classes as your programs are running? He explains how to do it. Most of the time, you'll want to steer clear of the techniques he describes, but sometimes they provide just the solution you need for a tricky problem you're facing. Furthermore, it's illuminating just to see what kinds of things can be done with C++. This book may frighten you, it may dazzle you, but when you've read it, you'll never look at C++ the same way again.

If you have anything to do with the design and implementation of C++ libraries, you would be foolhardy to overlook

- *Designing and Coding Reusable C++*, Martin D. Carroll and Margaret A. Ellis, Addison-Wesley, 1995, ISBN 0-201-51284-X.

Carroll and Ellis discuss many practical aspects of library design and implementation that are simply ignored by everybody else. Good libraries are small, fast, extensible, easily upgraded, graceful during template instantiation, powerful, and robust. It is not possible to optimize for each of these attributes, so one must make trade-offs that improve some aspects of a library at the expense of others. *Designing and Coding Reusable C++* examines these trade-offs and offers down-to-earth advice on how to go about making them.

Regardless of whether you write software for scientific and engineering applications, you owe yourself a look at

- *Scientific and Engineering C++*, John J. Barton and Lee R. Nackman, Addison-Wesley, 1994, ISBN 0-201-53393-6.

The first part of the book explains C++ for FORTRAN programmers (now *there's* an unenviable task), but the latter parts cover techniques that are relevant in virtually any domain. The extensive material on templates is close to revolutionary; it's probably the most advanced that's currently available, and I suspect that when you've seen the miracles these authors perform with templates, you'll never again think of them as little more than souped-up macros.

Finally, the emerging discipline of *patterns* in object-oriented software development (see page 123) is described in

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2.

This book provides an overview of the ideas behind patterns, but its primary contribution is a catalogue of 23 fundamental patterns that are useful in many application areas. A stroll through these pages will almost surely reveal a pattern you've had to invent yourself at one time or another,

and when you find one, you're almost certain to discover that the design in the book is superior to the ad-hoc approach you came up with. The names of the patterns here have already become part of an emerging vocabulary for object-oriented design; failure to know these names may soon be hazardous to your ability to communicate with your colleagues. A particular strength of the book is its emphasis on designing and implementing software so that future evolution is gracefully accommodated (see Items 32 and 33).

Design Patterns is also available as a CD-ROM:

- *Design Patterns CD: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8.

Magazines

For hard-core C++ programmers, there's really only one game in town:

- *C++ Report*, SIGS Publications, New York, NY.

The magazine has made a conscious decision to move away from its "C++ only" roots, but the increased coverage of domain- and system-specific programming issues is worthwhile in its own right, and the material on C++, if occasionally a bit off the deep end, continues to be the best available.

If you're more comfortable with C than with C++, or if you find the *C++ Report's* material too extreme to be useful, you may find the articles in this magazine more to your taste:

- *C/C++ Users Journal*, Miller Freeman, Inc., Lawrence, KS.

As the name suggests, this covers both C and C++. The articles on C++ tend to assume a weaker background than those in the *C++ Report*. In addition, the editorial staff keeps a tighter rein on its authors than does the *Report*, so the material in the magazine tends to be relatively mainstream. This helps filter out ideas on the lunatic fringe, but it also limits your exposure to techniques that are truly cutting-edge.

Usenet Newsgroups

Three Usenet newsgroups are devoted to C++. The general-purpose anything-goes newsgroup is `comp.lang.c++` . The postings there run the gamut from detailed explanations of advanced programming techniques to rants and raves by those who love or hate C++ to undergraduates the world over asking for help with the homework assignments they neglected until too late. Volume in the newsgroup is extremely high. Unless you have hours of free time on your hands, you'll want to employ a filter to help separate the wheat from the chaff. Get a good filter — there's a lot of chaff.

In November 1995, a moderated version of `comp.lang.c++` was created. Named `comp.lang.c++.moderated`, this newsgroup is also designed for general discussion of C++ and related issues, but the moderators aim to weed out implementation-specific questions and comments, questions covered in the extensive on-line FAQ ("Frequently Asked Questions" list),

flame wars, and other matters of little interest to most C++ practitioners.

A more narrowly focused newsgroup is ◦`comp.std.c++`, which is devoted to a discussion of ◦the C++ standard itself. Language lawyers abound in this group, but it's a good place to turn if your picky questions about C++ go unanswered in the references otherwise available to you. The newsgroup is moderated, so the signal-to-noise ratio is quite good; you won't see any pleas for homework assistance here.

Back to Item 35: : Familiarize yourself with the language standard
Continue to An `auto_ptr` Implementation

Back to Recommended Reading

An `auto_ptr` Implementation

Items 9, 10, 26, 31 and 32 attest to the remarkable utility of the `auto_ptr` template. Unfortunately, few compilers currently ship with a "correct" implementation.[1] Items 9 and 28 sketch how you might write one yourself, but it's nice to have more than a sketch when embarking on real-world projects.

Below are two presentations of an implementation for `auto_ptr`. The first presentation documents the class interface and implements all the member functions outside the class definition. The second implements each member function within the class definition. Stylistically, the second presentation is inferior to the first, because it fails to separate the class interface from its implementation. However, `auto_ptr` yields simple classes, and the second presentation brings that out much more clearly than does the first.

Here is `auto_ptr` with its interface documented:

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0);              // see Item 5 for a
                                            // description of "explicit"

  template<class U>                         // copy constructor member
  auto_ptr(auto_ptr<U>& rhs);               // template (see Item 28):
                                            // initialize a new auto_ptr
                                            // with any compatible
                                            // auto_ptr

  ~auto_ptr();

  template<class U>                         // assignment operator
  auto_ptr<T>&                              // member template (see
  operator=(auto_ptr<U>& rhs);              // Item 28): assign from any
                                            // compatible auto_ptr

  T& operator*() const;                     // see Item 28
  T* operator->() const;                    // see Item 28

  T* get() const;                           // return value of current
                                            // dumb pointer

  T* release();                             // relinquish ownership of
                                            // current dumb pointer and
                                            // return its value

  void reset(T *p = 0);                     // delete owned pointer;
                                            // assume ownership of p
private:
```

```
  T *pointee;

template<class U>                           // make all auto_ptr classes
friend class auto_ptr<U>;                   // friends of one another
};

template<class T>
inline auto_ptr<T>::auto_ptr(T *p)
: pointee(p)
{}

template<class T>
  inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
  : pointee(rhs.release())
  {}

template<class T>
inline auto_ptr<T>::~auto_ptr()
{ delete pointee; }

template<class T>
  template<class U>
  inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

template<class T>
inline T& auto_ptr<T>::operator*() const
{ return *pointee; }

template<class T>
inline T* auto_ptr<T>::operator->() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::get() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::release()
{
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}

template<class T>
inline void auto_ptr<T>::reset(T *p)
{
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

Here is `auto_ptr` with all the functions defined in the class definition. As you can see, there's no brain surgery going on here:

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}

  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}

  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

  T& operator*() const { return *pointee; }

  T* operator->() const { return pointee; }

  T* get() const { return pointee; }

  T* release()
  {
    T *oldPointee = pointee;
    pointee = 0;
    return oldPointee;
  }

  void reset(T *p = 0)
  {
    if (pointee != p) {
      delete pointee;
      pointee = p;
    }
  }
  private:
    T *pointee;

  template<class U> friend class auto_ptr<U>;
  };
```

If your compilers don't yet support `explicit`, you may safely `#define` it out of existence:

```
#define explicit
```

This won't make `auto_ptr` any less functional, but it will render it slightly less safe. For details, see Item 5.

If your compilers lack support for member templates, you can use the non-template `auto_ptr` copy constructor and assignment operator described in Item 28. This will make your `auto_ptrs` less convenient to use, but there is, alas, no way to approximate the behavior of member templates. If member templates (or other language features, for that matter) are important to you, let your compiler vendors know. The more customers ask for new language features, the sooner vendors will implement them.

Back to Recommended Reading

---

[1] This is primarily because the specification for `auto_ptr` as for years been a moving target. The final specification was adopted only in November 1997. For details, consult the `auto_ptr` information at this book's WWW Site. Note that the `auto_ptr` described here omits a few details present in the official version, such as the fact that `auto_ptr` is in the `std` namespace (see Item 35) and that its member functions promise not to throw exceptions.
Return